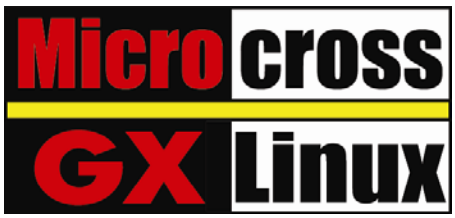


Embedded Linux Made Easy

Microcross GX-Linux™

Professional Embedded Linux Development Solution

User Guide

Version 1.0e-20051015

© 2005 Microcross, Inc.
Licensed Materials

MICROCROSS PRESS

Powered by GNU X-Tools™, SlickEdit® & Linux OS

ISBN: N/A

Microcross, Inc.
104 Borders Way, Suite 100,
Warner Robins, GA 31088

© 2005 Microcross, Inc.

All rights reserved.

This document and the GX-Linux™ Embedded Linux BSP described in it are furnished under license with Microcross, and as such, may be used or copied only in accordance with the terms of the license. The contents of this document are furnished for informational purposes only and are subject to change without notice. Microcross assumes no responsibility or liability for any errors or inaccuracies that may exist in this document. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, without the prior written consent of Microcross.

Trademarks

GX-Linux™, Visual X-Tools™ and GNU X-Tools™, *MICROCROSS PRESS*™ and Microcross® are all trademarks of Microcross, Inc.. All other brand and product names, services names, trademarks and copyrights are the property of their respective owners. This documentation contains copyright materials and has been prepared by Microcross Technical Publications; contact the Microcross Technical Publications staff for more information: **support@microcross.com**.

Disclaimer

Microcross, Inc. makes no representations or warranties with respect to the contents or use of this user guide, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Microcross, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes. Microcross, Inc. makes no representations or warranties with respect to any Microcross software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Microcross, Inc. reserves the right to make changes to any and all parts of Microcross software, at any time, without any obligation to notify any person or entity of such changes.

GX-Linux™ User Guide
Supports Product Part Numbers: MGXL400Lx-xxx, October 2005
Printed in USA

MICROCROSS PRESS

Table of Contents

Section 1.0 Overview and System Requirements 7

1.1 Overview 7
1.2 Components of the GX-Linux Standard Platform 7
1.3 Components of the GX-Linux Professional Platform 8

Section 2.0 Installation and Setup 9

2.1 Standard Platform Installation and Setup on a Linux Host Workstation 10
2.2 Professional Platform Installation and Setup on a Linux Host Workstation 13
2.3 Application Development Platform Installation and Setup on Windows Host Workstation 16
2.4 Uninstall GX-Linux 17
2.5 Uninstall GNU X-Tools Toolsuite 17

Section 3.0 Getting Started 18

3.1 Getting Started with the Standard Platform Linux Host 19
3.2 Getting Started with the Professional Platform on Linux Host 23
3.3 Getting Started with Application Development Platform on Windows Host 27
3.4 MicroMonitor Boot ROM Familiarity and Tips 31
3.4.1 Manual TFTP Up-Load Procedure 31
3.4.2 Other Useful MicroMonitor Commands 32

Section 4.0 Visual X-Tools Example Application Project 33

4.1 Hello Example Program 33

Section 5.0 Making Modifications to GX-Linux 35

5.1 Modifying GX-Linux Standard Platform 35
5.1.1 Building Your Own Applications 35
5.2 Modifying GX-Linux Professional Platform 36
5.2.1 GX-Linux BSP Components 36
5.2.2 Building All of the Components 36
5.2.3 Building All Components Excluding Examples 36
5.2.4 Building Your Own Applications 36
5.3 Changing the Kernel Configuration 37
5.3.1 Configuration Files 37
5.3.2 Changing the Working Configuration 37
5.3.3 Resetting the Kernel to the Default Configuration 38
5.3.4 Customizing the Kernel Configuration 38
5.4 Changing the Target File System 38
5.4.1 Adding Content to the Target File System 38

5.5 Putting the New System on the Target Hardware 39
5.5.1 Putting the New System on the Target Hardware 39
5.6 Read-Only and Read-Write File Systems 39
5.6.1 ROMFS File System 39
5.6.2 NFS Mounted Root Filesystem 39
5.6.3 Writing to On-Board Flash via a Pendrive (USB) 41
5.6.4 How to Prepare a Pendrive with a Root Filesystem and Work with GX-Linux 41
5.6.5 Writing to On-Board Flash via MMC/SD Card 42
5.6.6 Writing to On-Board Flash via a Compact Flash Card 42

Section 6.0 Debugging Applications 43

6.1 The GDB Debugger 43
6.1.1 GDB Debugger 43
6.1.2 gdbserver 43
6.1.3 gdbserver Communications 43
6.1.4 Graphical Front-Ends 43
6.1.5 Debug-able Files 44
6.1.6 Visual GDB 44
6.2 Debugging Examples 44
6.2.1 Example with Visual GDB 44
6.2.2 Example with GDB 44
6.3 Debugging Using the gdbinit 46

7.0 Networking 47

7.1 Default Kernel Configuration with Networking Enabled 47
7.2 Network Configuration 47

8.0 Serial Port 49

8.1 Serial Devices 49
8.1.1 Introduction 49
8.1.2 Default Serial Driver Configuration 49
8.2 Program Access to /dev/ttyS0 49
8.3 System Console 49
8.3.1 Default Console 49
8.3.2 Login via the Serial Port 49
8.3.3 Standard IO 49
8.3.4 Restrictions 50
8.4 Serial Settings 50
8.5 Kernel Console over Serial 50

Section 9. How to Use GNU X-Tools 51

9.1 Introduction 51
9.2 Using the GNU X-Tools Command Line Tool `(xtools)` 51
9.3 Invoking GNU X-Tools 52
9.4 Using the GNU X-Tools Toolsuite 53

9.4.1 Simple Example 53
9.4.2 GNU X-Tools Toolsuite Description 54
9.4.3 Control Program (GCC) 55
9.5 Controlling the Tools Using the GCC 58
9.5.1 GCC Options Commonly Used 59
9.6 Controlling Linker from GCC 61
9.7 Compilation Options 62
9.7.1 Displaying compiler behavior 62
9.7.2 C Language Options 62
9.7.3 Preprocessor Options 63
9.7.4 Options to Specify Libraries, Paths and Startup Files 63
9.7.5 Debugging and Profiling Options 65
9.7.6 Optimization 66
9.7.7 Passing Options to the Assembler or Linker 67
9.8 Using the GNU Assembler 67
9.9 Using the Linker 70
9.9.1 Invoking ld 70
9.9.2 Linker Options 71
9.9.3 Linker Scripts 72
9.9.4 Link-Order Optimization 73
9.9.5 The C Runtime (crt0) 73
9.10 Object Translation (ELF to Binary, SREC, etc.) 73
9.11 Creating/Updating Libraries 74
9.12 GNU Libraries 75
Appendix 1. GNU X-Tools ARM Linux Toolsuite 76
Appendix 2. GNU X-Tools MIPS Linux Toolsuite 84
Appendix 3. GNU X-Tools PowerPC Linux Toolsuite 92
Appendix 4. Common Minicom Problems 101
Appendix 5. Glossary 102
Appendix 6. Additional Resources 104

Section 1.0 Overview and System Requirements

1.1 Overview

The GX-Linux User Guide for building Embedded Linux applications has important tutorial information on how to use the BSP and tools effectively and jump start the learning process so you can focus on developing your application. Specifically, this User Guide provides information on how to get the GX-Linux setup with GNU X-Tools and GX-Linux Board Support Package (BSP) and building applications for the target board. Two versions of GX-Linux are offered: Standard Platform and Professional Platform. There are differing requirements for each Platform.

System Requirements for GX-Linux Software BSP Installation

1. Hardware:
   Linux OS and Host with an x86 or Intel® Pentium® family processor
   keyboard
   mouse,
   video monitor

2. Linux distribution pre-installed on a host computer:
   Red Hat® Linux 7.x, 8.0, 9.0, EWS v3.0 or newer (complete install)
   Debian (complete install)
   Linspire® Linux v4.5 Developer's Edition

3. CPU clock rate: 400 MHz or higher i586/i686 PC

4. Recommended system RAM: 256 MB

5. Hard disk space required: 1 GB

6. A network card installed and configured on the host computer

7. A Cogent OEM board with breakout board and power supply

8. RS-232 serial port (female-to-female, 9-pin, NULL modem cross-over or a hub for Ethernet connection)

9. Ethernet cross-over cable for standalone configuration or two standard UTP patch cables for a LAN / Workgroup configuration.

1.2 Components of the GX-Linux Standard Platform

The GX-Linux Standard Platform package includes a single compact disk (CD-ROM) loaded with GNU X-Tools, documentation, GX-Linux BSP binaries with example programs, and boot ROM recovery files for the Cogent uSBC. Go through the installation and initial setup procedures in the next section, and then go to Section 3 to get started.

- Cogent uSBC OEM board and breakout board with power supply, null modem serial cable, and cross-over Ethernet cable
- GNU X-Tools Linux toolsuite and ELF toolsuite
- uMON bootloader and GX-Linux startup script (startlinux)
- Linux kernel v2.6.x binary for the target processor (pre-built)
- uClibc Library (pre-built) – a small C library for Embedded Linux users
- Busybox on a ROM filesystem with basic system utilities and example applications (pre-built)
- GX-Linux Standard Makefile for building examples applications and filesystem

1.3 Components of the GX-Linux Professional Platform

The GX-Linux Professional Platform package includes a compact disk (CD-ROM) loaded with GNU X-Tools, documentation, GX-Linux BSP sources, build scripts with example programs, and boot ROM recovery files for the Cogent uSBC, plus a second CD-ROM with Visual X-Tools IDE. Consult Visual X-Tools User Guide for installation instructions. Go through the installation and initial setup procedures in the next section, and then go to Section 3 to get started.

- Cogent uSBC OEM board and breakout board with power supply, null modem serial cable, and Ethernet cable
- GNU X-Tools Linux toolsuite and ELF toolsuite
- Visual X-Tools Integrated Development Environment
- uMON bootloader and GX-Linux startup script (startlinux)
- Linux kernel v2.6.x binary for the target processor (pre-built)
- uClibc Library – a small C library for Embedded Linux users
- Busybox on a ROM filesystem with basic system utilities and example applications
- All GX-Linux component Makefile build scripts for rebuilding binary builds for Linux kernel, uClibc library, BSP, example applications and filesystem

Section 2.0 Installation and Setup

This section covers a number of procedures to install all components of the complete Microcross GX-Linux solution. Section 2.4 shows how to install and setup an application development platform that can be used either for the Standard or professional platforms on a Windows host workstation. The limitation to this is that a developer cannot modify and rebuild the kernel on a Windows host workstation. At present, the GX-Linux kernel can only be rebuilt (out-of-the-box) on a Professional Platform. The zImage must then be copied into the `'gxlinux'` build directory on the Windows hosted Application Development Platform.

Section 2.1
Standard Platform Installation and Setup on a Linux Host Workstation

Section 2.2
Professional Platform Installation and Setup on a Linux Host Workstation

Section 2.3
Application Development Setup on Windows -- Available for Standard and Professional Platforms

Section 2.4
Uninstall GX-Linux

Section 2.5
Uninstall GNU X-Tools Toolsuite

Note: Wherever you see (cr), it means Carriage Return (i.e., `'Enter'` on the keyboard).

2.1 Standard Platform Installation and Setup on a Linux Host Workstation

The installation procedure for the Standard Platform is a command line procedure and is simple and straightforward. We assume that all requirements in Section 1 are met and that you have a Linux OS running on your workstation or server before getting started with this installation. All commands are done from the Bash/xterm shell in Linux.

Step1.

Insert the GX-Linux distribution CD into your host computer's CD-ROM drive.

Step 2.

Start a Bash/xterm shell and login as `'root'` (or use `'su'`). You will need to do this each time you install or uninstall GNU X-Tools toolsuite because you will need write-access to the `'/usr/bin'` and `'/usr/lib'` directories where the toolsuite is installed.

Step 3.

Mount the GX-Linux distribution CD on the CD-ROM drive using the `'mount'` command,

Example:

```
# mount -t iso9660 /dev/cdrom /mnt/cdrom  (cr)
```

The system will announce that it has mounted the CD-ROM as read-only. **Note:** On some Linux systems, the CD-ROM will be mounted by an automounter, which means using mount in step 3 is not required, or the procedure may need to be modified to accommodate non-standard Linux distributions or a machine with multiple CD-ROM drives that have differing naming conventions.

Step 4.

Install the GNU X-Tools command tool for the Bash/xterm shell by executing the following commands:

```
# cd /  (cr)
# tar -xvzf /mnt/cdrom/bin/rcfiles.tgz  (cr)
```

At this point you should be able to execute the `'xtools'` command tool inside of the Bash shell by its name. The command, `'xtools' (cr),` should invoke the abbreviated tool help screen that shows the basic operating commands (see Section 10 for a complete description of the command tool).

Step 5.

To install the two GNU X-Tools toolsuites from the GX-Linux CD, simply substitute the target alias (i.e., <target-alias> = arm-linux, mips-linux, mipsel-linux, ppc-linux, arm-elf, mips-elf, etc.) and issue the command:

```
#  xtools install <target-alias> (cr) (for the first toolsuite)
#  xtools install <target-alias> (cr) (for the second toolsuite)
```

These commands will extract the toolsuite tarballs, installing all files in their appropriate locations. The install command assumes the default archive source path is `'/mnt/cdrom/bin'` using `'/mnt/cdrom'` as the CD-ROM mount point. If your default mount point is different, you can change the default value by editing

'/usr/bin/xtools' file. If you would like to override the default path to the distribution medium, you may enter the command as 'xtools install arm-linux <arc-path>'. The '<arc-path>' should always end in 'bin' as the tar archives are in the bin directory on the CD-ROM.

```
e.g.:
# xtools install arm-linux /mnt/cdrom/bin
# xtools install arm-elf /mnt/cdrom/bin
```

Table 2.1 shows where the installation directories are located. The '/usr/<target-alias>' directory has two important subdirectories ('lib' and 'include') that contain the runtime libraries and header files. If you rebuild uClibc library, these are the directories where it gets installed along with the kernel headers.

Table 2.1 GNU X-Tools Installation Directories

Directory	Description
/usr/bin	Front-end binaries for the development tools, like the compiler, assembler, utilities, and debugger. The tools run on the Linux host workstation and operate on files destined for the target hardware.
/usr/<target-alias>	Development tool binaries include files, and libraries specific to the version of Linux being used.
/usr/lib/gcc-lib/<target-alias>	Compiler library directory.

Step 6.

If you do not have the 'tcl8.4/tk8.4' library installed on your Linux OS, you may need to install the correct version that Microcross ships on the GX-Linux distribution CD. You can also tryout your TCL/TK configuration by issuing the command '<target-alias>-gdbtk' and observe the start of the program – if there are any errors, then you know that there is a mis-match in configuration.

Also, you can check your TCL/TK version by issuing the following commands from an xterm/Bash shell:

```
$ tclsh (cr)
% info tclver (cr)  (version is echoed back)
% exit (cr)
```

If you need to install the right version of TCL/TK, the following is the command to issue to install the library from Microcross' CD-ROM:

```
- login as 'su' or root if not already done so
# xtools install share-gdbtcl (cr)
# exit (cr)
```

You should now be logged out of 'su/root'. Note if you had any errors in the last step due to 'xtools' command script not finding the proper mount path, then simply add the path after 'share-gdbtcl' (e.g., 'xtools install share-gdbtcl /mnt/cdrom/bin').

Step 7.

Select or create an *install* directory on your Linux host machine, preferably in your '/home/<user-name>' directory or a directory where you have full permissions. We call this the *install* directory because the *build* directory will be untarred to this directory. The *build* directory is named 'gxlinux', and when you untar the GX-

Linux BSP tarfile, it creates this build directory at the point of install. The Build contents for GX-Linux Standard Platform are shown in Table 2.2.

Table 2.2 Standard Platform Build Directory

Directory / File Name	Description of 'gxlinux'' Standard Platform Build directory
config	Default Linux target/board configuration files.
examples	Directory of 22 example applications that are built and copied into the root filesystem (/usr/bin).
linux	Directory of the pre-built kernel zImage file.
Makefile	Master makefile for Standard Platform
rootfs	Directory of root filesystem
system.cfg	File created at make time that stores configuration settings (BOARD and ARCH).
varetc	Directory contains the filesystem etc and var directory contents; the etc directory contains the system configuration files for the host, and the var directory contains logs, lock files and process specific data files.

Table 2.3 shows the *source* directory contents with all of the source files that come with GX-Linux. The only file that is needed to install, however, is 'gxlinux-<ver>-<csbxxx>-bin.tar.gz' and the next step shows how to install this tarfile.

Table 2.3 Source Directory (src) on CD-ROM

Source File Name	Description
busybox-<ver>.tar.gz	Tar file contains the BusyBox source tree.
config.tar.gz	Tar file contains the configuration files for the BSP.
examples.tar.gz	Tar file contains 20 example programs that get built and added to '/usr/bin' dir in the ROM filesystem.
gxlinux-<ver>-<csbxxx>-bin.tar.gz	GX-Linux Standard Platform BSP with Build directory
linux-<ver>-csbxxx-<date>.tar.gz	Tar file of the Kernel source tree.
rootfs.tar.gz	Tar file of the root filesystem.
uClibc-<ver>.tar.gz	Tar file of uClibc static and shared runtime libraries.
varetc.tar.gz	Tar file of filesystem elements that include the 'etc' and 'var' dirs.

Step 8.

Browse the CD-ROM 'src' directory to get the complete filename (format is 'gxlinux-<ver>-<csbxxx>-bin.tar.gz').

Example: Installing GX-Linux BSP

- Open an xterm/Bash shell.
- Mount the GX-Linux CD-ROM if it is not already mounted.
- Issue the following commands to create a build directory:

```
$ cd /home/<user-name>  (cr) (other other install directory)
$ tar xvzf /mnt/cdrom/src/gxlinux-0.1-csb637-bin.tar.gz (cr)
```

Note: You might need to substitute the right path to the CD-ROM if your system differs from the standard CD-ROM mount pathing. The build directory gets created at the point you install the GX-Linux BSP and is named 'gxlinux'.

Step 9.

At this point the BSP and toolsuite are installed. Next, go to Section 3 to configure the target board serial interface, network settings, and then run GX-Linux and run example applications on the Cogent uSBC.

2.2 Professional Platform Installation and Setup on a Linux Host Workstation

The installation procedure for the Professional Platform is a command line procedure and is simple and straightforward. We assume that all requirements in Section 1 are met and that you have a Linux OS running on your workstation or server before getting started with this installation. All commands are done from the Bash/xterm shell in Linux.

Step1.

Insert the GX-Linux distribution CD into your host computer's CD-ROM drive.

Step 2.

Start a Bash/xterm shell and login as 'root' (or use 'su'). You will need to do this each time you install or uninstall GNU X-Tools toolsuite because you will need write-access to the '/usr/bin' and '/usr/lib' directories where the toolsuite is installed.

Step 3.

Mount the GX-Linux distribution CD on the CD-ROM drive using the 'mount' command,

Example:

```
# mount -t iso9660 /dev/cdrom /mnt/cdrom  (cr)
```

The system will announce that it has mounted the CD-ROM as read-only. **Note:** On some Linux systems, the CD-ROM will be mounted by an automounter, which means using mount in step 3 is not required, or the procedure may need to be modified to accommodate non-standard Linux distributions or a machine with multiple CD-ROM drives that have differing naming conventions.

Step 4.

Install the GNU X-Tools command tool for the Bash/xterm shell by executing the following commands:

```
# cd /  (cr)
# tar -xvzf /mnt/cdrom/bin/rcfiles.tgz  (cr)
```

At this point you should be able to execute the 'xtools' command tool inside of the Bash shell by its name. The command, 'xtools' (cr), should invoke the abbreviated tool help screen that shows the basic operating commands (see Section 10 for a complete description of the command tool).

Step 5.

To install the two GNU X-Tools toolsuites from the GX-Linux CD, simply substitute the target alias (i.e., <target-alias> = arm-linux, mips-linux, mipsel-linux, ppc-linux, arm-elf, mips-elf, etc.) and issue the command:

```
#  xtools install <target-alias> (cr) (for the first toolsuite)
#  xtools install <target-alias> (cr) (for the second toolsuite)
```

These commands will extract the toolsuite tarballs, installing all files in their appropriate locations. The install command assumes the default archive source path is '/mnt/cdrom/bin' using '/mnt/cdrom' as the CD-ROM mount point. If your default mount point is different, you can change the default value by editing '/usr/bin/xtools' file. If you would like to override the default path to the distribution medium, you may enter the command as 'xtools install arm-linux <arc-path>'. The '<arc-path>' should always end in 'bin' as the tar archives are in the bin directory on the CD-ROM.

e.g.:

```
# xtools install arm-linux /mnt/cdrom/bin
# xtools install arm-elf /mnt/cdrom/bin
```

Table 2.4 shows where the installation directories are located. The '/usr/<target-alias>' directory has two important subdirectories ('lib' and 'include') that contain the runtime libraries and header files. If you rebuild uClibc library, these are the directories where it gets installed along with the kernel headers.

Table 2.4 GNU X-Tools Installation Directories

Directory	Description
/usr/bin	Front-end binaries for the development tools, like the compiler, assembler, utilities, and debugger. The tools run on the Linux host workstation and operate on files destined for the target hardware.
/usr/<target-alias>	Development tool binaries include files, and libraries specific to the version of Linux being used.
/usr/lib/gcc-lib/<target-alias>	Compiler library directory.

Step 6.

If you do not have the 'tcl8.4/tk8.4' library installed on your Linux OS, you may need to install the correct version that Microcross ships on the GX-Linux distribution CD. You can also tryout your TCL/TK configuration by issuing the command '<target-alias>-gdbtk' and observe the start of the program – if there are any errors, then you know that there is a mis-match in configuration.

Also, you can check your TCL/TK version by issuing the following commands from an xterm/Bash shell:

```
$ tclsh (cr)
% info tclver (cr)  (version is echoed back)
% exit (cr)
```

If you need to install the right version of TCL/TK, the following is the command to issue to install the library from Microcross' CD-ROM:

```
- login as 'su' or root if not already done so
# xtools install share-gdbtcl (cr)
# exit (cr)
```

You should now be logged out of 'su/root'. Note if you had any errors in the last step due to 'xtools' command script not finding the proper mount path, then simply add the path after 'share-gdbtcl' (e.g., 'xtools install share-gdbtcl /mnt/cdrom/bin').

Step 7.

Select or create an *install* directory on your Linux host machine, preferably in your '/home/<user-name>' directory or a directory where you have full permissions. We call this the *install* directory because the *build* directory will be untarred to this directory. The *build* directory is named 'gxlinux', and when you untar the GX-Linux BSP tarfile, it creates this build directory at the point of install. The Build contents for GX-Linux Professional Platform are shown in Table 2.5.

Table 2.5 Professional Platform Build Directory

Directory / File Name	Description of 'gxlinux'' Professional Platform Build directory
busybox	Directory of BusyBox utilities
config	Default Linux target configuration files.
examples	Directory of 22 example applications that are built and copied into the root filesystem (/usr/bin).
linux	Directory of the kernel source tree.
Makefile	Master makefile
rootfs	Directory of root filesystem
system.cfg	File created at make time that stores configuration settings (BOARD and ARCH).
uClibc	Directory of uClibc C and math shared and static library sources
varetc	Directory contains the filesystem etc and var directory contents; the etc directory contains the system configuration files for the host, and the var directory contains logs, lock files and process specific data files.

Table 2.6 shows the *source* directory contents with all of the source files come with GX-Linux. The only file that is needed to install, however, is 'gxlinux-<ver>-<csbxxx>-bin.tar.gz' and the next step shows how to install this tarfile.

Table 2.6 Source Directory (src) on CD-ROM

Source File Name	Description
busybox-<ver>.tar.gz	Tar file contains the BusyBox source tree.
config.tar.gz	Tar file contains all of the configuration files for the BSP.
examples.tar.gz	Tar file contains 20 example programs that get built and added to '/usr/bin' dir in the ROM filesystem.
gxlinux-<ver>-<csbxxx>-<date>.tar.gz	GX-Linux Professional Platform BSP with Build directory
linux-<ver>-csbxxx-<date>.tar.gz	Tar file of the Kernel source tree.
rootfs.tar.gz	Tar file of the root filesystem.
uClibc-<ver>.tar.gz	Tar file of uClibc static and shared runtime libraries.
varetc.tar.gz	Tar file of filesystem elements that include the 'etc' and 'var' dirs.

Step 8.

Browse the CD-ROM 'src' directory to get the complete filename (format is 'gxlinux-<ver>-<csbxxx>-<date>.tar.gz').

Example with notional names: Installing GX-Linux BSP

- Open an xterm/Bash shell.
- Mount the GX-Linux CD-ROM if it is not already mounted.
- Issue the following commands to create a build directory:

```
$ cd /home/<user-name>  (cr) (other other install directory)
$ tar xvzf /mnt/cdrom/src/gxlinux-1.0e-csb637-20051015.tar.gz (cr)
```

Note: You might need to substitute the right path to the CD-ROM if your system differs from the standard CD-ROM mount pathing. The build directory gets created at the point you install the GX-Linux BSP and is named `'gxlinux'`.

Step 9.

Now, you need to set your user permissions on the host Linux operating system so you can write to `'/usr/<target-alias/lib'` and `'/usr/<target-alias>/include'` directories. Ask your system administrator to add you to the `'bin','users', and 'wheel'` groups. An example for Red Hat v9.0 follows.

Example
Click on `'Start|System Settings|Users and Groups'`. Enter your root password. Double click on the username that you wish to add permissions. The `'User Permissions'` dialog box pops up. Click on the `'Groups'` tab and select the `'bin','users',` and `'wheel'` groups check boxes and click `'OK'`. Click on `'File|Quit'` to exit.

At this point the BSP and toolsuite are installed and permissions are set. Next, go to Section 3 to configure the target board serial interface, network settings, and then run GX-Linux and example applications on the Cogent uSBC.

2.3 Application Development Setup on Windows Host – Available for Standard and Professional Platforms

The Application Development Platform is provided for Windows users who want to create their Linux applications in a more familiar environment on Windows using Cygwin and Visual X-Tools. The limitation is that you will not be able to re-build the kernel or uClibc on Windows/Cygwin host; however, combined with the Professional Platform, the user can recreate the kernel and custom libraries and copy them into the GX-Linux environment on Windows/Cygwin.

Files in the `'win'` directory on the GX-Linux CD-ROM include:

- 0x0409.ini
- Autorun.inf
- Cygwin with GX-Linux Platform CSBxxx.msi
- Data1.cab
- instmsia.exe
- instmsiw.exe
- setup.exe -- Setup utility to install Cygwin & GX-Linux Application Setup on Windows
- setup.ini

Open Explorer and navigate to the `'win'` directory on the GX-Linux CD-ROM and execute setup. Follow the on-screen instructions to complete the installation. The installation will take approximately 20 minutes or more

depending on the speed of your computer. Cygwin is a Linux/Unix platform itself, so users can build, run, and test applications using a familiar Windows environment.

If you have the capacity to rebuild the kernel to add custom settings and drivers, the do so on a Linux host machine and copy the image (it may be called 'zImage' or 'vmlinux')

Next, go to Section 3 to configure the target board serial interface, network settings, and then run GX-Linux and example applications on the Cogent uSBC.

2.4 Uninstall GX-Linux

Linux Host
To remove a previously installed version of the GX-Linux BSP software, issue the following command. WARNING: Before removing the tree, be sure to backup any files you want to keep.

```
$ cd /<peer-dir-to-gxlinux>    (cr)
$ rm -rf gxlinux               (cr)
```

Windows Host
To uninstall the Application Development Platform, click on the 'Start|Control Panel' and select 'Add/Remove Programs'. Select Remove Cygwin with GX-Linux Platform. Manually delete the 'Cygwin' directory.

2.5 Uninstall GNU X-Tools Toolsuite

If at any time you need to uninstall GNU X-Tools GX-Linux Toolsuite, follow the two steps below.

1) Start xterm/Bash shell and login as 'su' or 'root'.
2) Issue the command 'xtools remove <target-alias>'

Target aliases for GNU X-Tools GX-Linux Toolsuites include 'arm-linux', 'mips-linux', 'mipsel-linux', 'ppc-linux', 'arm-elf, 'mips-elf', etc.. Note that on a Windows host workstation, the developer uninstalls the GNU X-Tools (on Cygwin) at the same time as GX-Linux is uninstalled.

Section 3.0 Getting Started

There are two configurations for GX-Linux and an optional Application Development Platform (comes with both the Standard and Professional Platforms). What this means is a developer can build applications on Windows/Cygwin using a cross-development environment that is identical to the Standard Platform build environment. Like the Standard Platform edition, the limitation is that you cannot rebuild the kernel and libraries on Windows– they are pre-built. With the Professional Platform, however, the developer has an option to build all, modify kernel and libraries, and then rebuild on a Linux host. Then, the developer may overlay the pre-built kernel and modified libraries onto the Windows/Cygwin environment and proceed to build applications in that environment. This option does not prevent the developer, however, from performing all tasks, including application development, on a Linux host, which he/she many prefer to do.

Below are the names of the appropriate sections for getting started with your Platform version. Section 3.4 summarizes useful commands and tips using MicroMonitor Boot ROM.

Section 3.1
Getting Started with the Standard Platform on Linux Host

Section 3.2
Getting Started with the Professional Platform on Linux Host

Section 3.3
Getting Started with Application Development Setup on Windows Host

Section 3.4
MicroMonitor Boot ROM Familiarity and Tips

3.1 Getting Started with the Standard Platform Linux Host

Board Connection and Initial GX-Linux Bootup
Make physical connections per the diagram layout in Figure 3.1. If connecting to a hub on a network, disregard connecting an Ethernet cross-over cable. To get the target board booted up and communicating with your host Linux computer, you will need to configure your serial port and MicroMonitor before loading GX-Linux binaries.

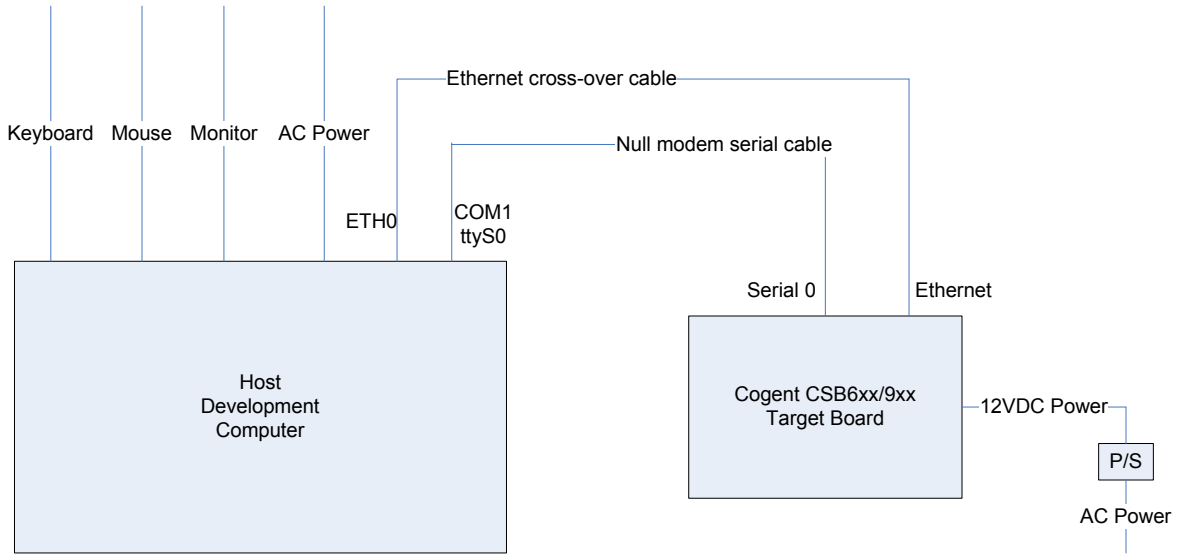
Step 1. Setup Serial Terminal

- If you do not have a serial terminal program running on your host machine, you will need to install one. We recommend a serial terminal program such as `'minicom'` or similar.
- Connect the Cogent provided null-modem serial cable to the target board and host computer.
- Configure your serial terminal program to connect to the target board using the following settings:

```
38400 baud
8 data bits
No parity
1 stop bit
no flow control
```

- Connect power supply to the board, and the serial terminal should show the MicroMonitor startup banner and the uMON> command prompt.

Figure 3.1 Diagram of Board Connections to Host Computer



Step 2. Setup uMON

You have two options to setup the network file, `'monrc'` in MicroMonitor: manual settings or DHCP/BootP. Consult your system administrator if you need help in determining your network settings.

Manual network settings: The settings shown are notional and are for reference of the procedure only. Use the uMON command line interface in your serial terminal session to enter these commands.

```
uMON> set -c
uMON> set IPADD 192.168.0.110    (IP address)
uMON> set NETMASK 255.255.255.0 (network mask)
uMON> set GIPADD 192.168.0.250   (gateway IP address)
uMON> set -f monrc
uMON> reset
```

DHCP/BOOTP network settings: Follow this example and enter the commands using your serial terminal uMON command line interface.

```
uMON> set -c
uMON> set IPADD DHCP (or DHCPv or DHCPV or BOOTP)
uMON> set -f monrc
uMON> reset
```

Select only one for IPADD setting

`DHCP`	No verbosity enabled
`DHCPv`	Limited verbosity enabled
`DHCPV`	Extreme verbosity enabled
`BOOTP`	Runs BootP

IMPORTANT: After configuring the network settings and initiating the reset command, the target board will re-boot MicroMonitor. Notice that in the serial terminal interface that the new IP address is displayed; if the new IP address does not show up on the display, go back to the previous make the necessary corrections to `'monrc'`. IMPORTANT: When using DHCP/BootP, the new IP address should be annotated in your notes, because you will need this IP address for loading GX-Linux files via TFTP. If your DHCP/BootP server assigns a different IP address every time you reboot MicroMonitor, you may want to consider manually setting the network IP address while in the development phase of your project, because the IP address is a key parameter to your TFTP communications between the host development platform and the target board.

Step 3. Modify Environment Variables

Open Visual X-Tools or another editor and edit the master `'Makefile'` in the *build* directory. The *build* directory is the `'gxlinux'` directory created at installation time. Edit configuration variables as follows:

`ARCH = <arch-of-target>`	(i.e., arm, mips, ppc, etc.)
`BOARD = <csbxxx>`	(if not set, enter the Cogent uSBC name – e.g., csb337, csb637, etc.)
`TGTIP = <IP-address>`	(enter the IP address assigned to the target board in the last step)

Optionally, you can modify the other configuration variables that apply, but the three most important are the ones above.

Step 4. Make All

From the *build* directory, `'gxlinux'`, make the example programs and create a ROM image for the `'romfs.img'` filesystem by issuing the following command:

```
$ make all    (cr)
```

(builds example applications, installs them, and re-generates a ROM filesystem)

Step 5. Load Binaries and Startup Script

This step assumes that the target board is powered up, connected to Ethernet, and uMON prompt is displayed in a serial terminal. From the build directory in xterm/Bash, issue a make command that will load the GX-Linux binaries and startup script (we assume you assigned the target board an IP address and edited the makefile, connections are made and the board is powered on).

```
$ make load        (cr)
```

Three images should start loading onto the board – see the serial terminal interface for upload response in MicroMonitor. The images loaded are as follows: 1) `'zImage'` (kernel); 2) `'romfs.img'` (ROM root filesystem); and 3) `'startlinux'` (startup script).

Step 6. Reset and Boot Linux

Press the board reset switch once the files have completely transferred. From the serial terminal window, issue the following command:

`uMON> startlinux (cr)` (this script calls the Linux kernel and filesystem to execute)

The Linux kernel should uncompress and mount a ROM filesystem. If there are any errors, reset the board and repeat this step. Occasionally the kernel aborts on a bootup due to a corrupted flash file system. Follow steps in the MicroMonitor tips section to issue a command (uMON> tfs clean) to defrag the flash file system.

The Linux startup script is set with a Tiny Filesystem attribute that makes `'startlinux'` an executable like a batch file in DOS. To change this attribute so that you can automatically boot with query into Linux with a query to stop after MicroMonitor boots up, issue the following command after boot up of MicroMonitor:

```
uMON> tfs -feB cp startlinux startlinux  (cr)
```

WARNING: `'-feB'` is case sensitive and if you use a lower case `'b'` in the switch option, you will not be able to boot into uMON again without re-flashing through the JTAG, which is complicated and time consuming. To change the attribute back to manual boot, substitute `'-fe'` in place of `'-feB'` in the above command above. To stop the automatic boot, press any key while uMON is booting and before Linux boots. The bootup will stop at the uMON command line prompt.

Step 7. Run Example Applications

When the GX-Linux boots to a Linux command shell prompt, you can enter the following commands in the serial terminal to get acquainted with all of the example programs loaded in the `'/usr/bin'`; each example file has a `'.x'` file extension. In the serial terminal command line interface window, issue the following commands:

```
# cd /usr/bin      (cr)
# ls               (cr) (shows all examples; to execute examples, perform the next step)
# hello.x          (cr) (executes hello world program)
```

There are approximately 20 other programs that can be executed in the same manner, and each filename ends in `'.x'` extension. Tryout as many of the examples as you wish. Read the next section to become more familiarized with MicroMonitor.

Step 8. How to Add Your Own Applications

The fastest way to get your own application on the target board is to copy your application – named with a '.x' file extension -- into the examples directory, which is under the build directory, and issue the following command in the xterm/Bash shell:

```
$ make install-apps romfs (cr)
$ make load               (cr)  (reload your images to the target board)
```

After uploading three files to the target board (startlinux, zImage, and romfs.img), you can manually start GX-Linux by issuing the following commands:

```
$ (cr)                  (get a command prompt after upload)
$ reset (cr)            (get the board in a refresh state)
$ startlinux (cr)       (Linux boots to a user prompt '#')
```

On the target board in the '/usr/bin' directory are a dozen example programs that can be run from the Minicom interface. Issue the following commands:

```
#  cd /usr/bin (cr)
#  ls (cr)
```

You can see all examples; they have a '.x' file extension on them. To run each example, issue the by-name command:

```
#  double.x (cr)
#  float.x  (cr)
#  hello.x  (cr)
etc…
```

Step 9. Useful Linux Commands

```
# cat /proc/cpuinfo
```
Show information such as bogomips, architecture, and manufacture information.

```
# reboot
```
Reboots system into uMON.

3.2 Getting Started with the Professional Platform on Linux Host

Board Connection and Initial GX-Linux Bootup
Make physical connections per the diagram layout in Figure 3.2. If connecting to a hub on a network, disregard connecting an Ethernet cross-over cable. To get the target board booted up and communicating with your host Linux computer, you will need to configure your serial port and MicroMonitor before loading GX-Linux binaries.

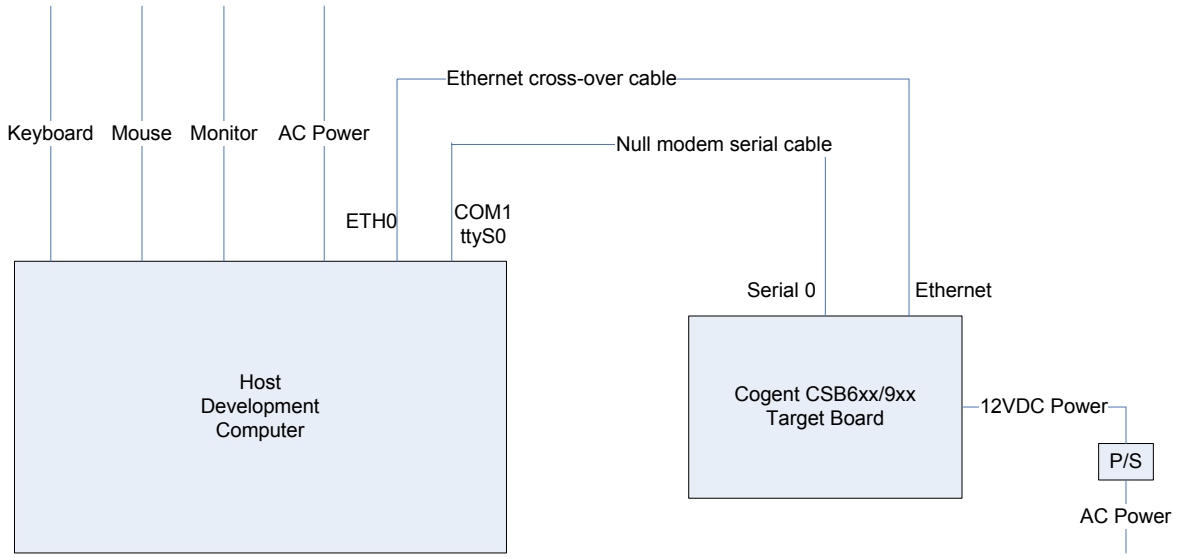
Step 1. Setup Serial Terminal

- If you do not have a serial terminal program running on your host machine, you will need to install one. We recommend a serial terminal program such as `'minicom'` or similar.
- Connect the Cogent provided null-modem serial cable to the target board and host computer.
- Configure your serial terminal program to connect to the target board using the following settings:

```
38400 baud
8 data bits
No parity
1 stop bit
no flow control
```

- Connect power supply to the board, and the serial terminal should show the MicroMonitor startup banner and the uMON> command prompt.

Figure 3.2 Diagram of Board Connections to Host Computer



Step 2. Setup uMON

You have two options to setup the network file, `'monrc'` in MicroMonitor: manual settings and DHCP/BootP. Consult your system administrator if you need help in determining your network settings.

Manual network settings: The settings shown are notional and are for reference of the procedure only. Use the uMON command line interface in your serial terminal session to enter these commands.

```
uMON> set -c
uMON> set IPADD 192.168.0.110    (IP address)
uMON> set NETMASK 255.255.255.0 (network mask)
uMON> set GIPADD 192.168.0.250   (gateway IP address)
uMON> set -f monrc
uMON> reset
```

DHCP/BOOTP network settings: Follow this example and enter the commands using your serial terminal uMON command line interface.

```
uMON> set -c
uMON> set IPADD DHCP (or DHCPv or DHCPV or BOOTP)
uMON> set -f monrc
uMON> reset
```

Select only one for IPADD setting

`DHCP`	No verbosity enabled
`DHCPv`	Limited verbosity enabled
`DHCPV`	Extreme verbosity enabled
`BOOTP`	Runs BootP

IMPORTANT: After configuring the network settings and initiating the reset command, the target board will re-boot MicroMonitor. Notice that in the serial terminal interface that the new IP address is displayed; if the new IP address does not show up on the display, go back to the previous make the necessary corrections to `'monrc'`. IMPORTANT: When using DHCP/BootP, the new IP address should be annotated in your notes, because you will need this IP address for loading GX-Linux files via TFTP. If your DHCP/BootP server assigns a different IP address every time you reboot MicroMonitor, you may want to consider manually setting the network IP address while in the development phase of your project, because the IP address is a key parameter to your TFTP communications between the host development platform and the target board.

Step 3. Modify Environment Variables

Open Visual X-Tools or another editor and edit the master `'Makefile'` in the *build* directory. The build directory is `'gxlinux'` directory created at installation time. Edit configuration variables as follows:

`ARCH = <arch-of-target>`	(i.e., arm, mips, ppc, etc.)
`BOARD = <csbxxx>`	(if not set, enter the Cogent uSBC name – e.g., csb637, csb625, etc.)
`TGTIP = <IP-address>`	(enter the IP address assigned to the target board in the last step)

Optionally, you can modify the other configuration variables that apply, but the three most important are the ones above.

Step 4. Make All

Start an xterm/Bash shell, change directory to the *build* directory and issue the following command:

`$ make all (cr)` (the system configures and builds everything for the target board)

If there are any build or install errors, make sure the toolsuite and BSP were installed correctly per instructions in Section 2. If you still have problems, contact Microcross support at +1.478.953.1907 or support@microcross.com

Step 5. Load Binaries and Startup Script

This step assumes that the target board is powered up, connected to Ethernet, and uMON prompt is displayed in a serial terminal. From the build directory in xterm/Bash, issue a make command that will load the GX-Linux binaries and startup script (we assume you assigned the target board an IP address and edited the makefile, connections are made and board is powered on).

```
$ make load        (cr)
```

Three images should start loading onto the board – see the serial terminal interface for upload responses in MicroMonitor. The images loaded are as follows: 1) `'zImage'` (kernel); 2) `'romfs.img'` (ROM root filesystem); and 3) `'startlinux'` (startup script).

Step 6. Reset and Boot Linux

Press the board reset switch once the files have completely transferred. From the serial terminal window, issue the following command:

```
uMON> startlinux (cr)
```
(this script calls the Linux kernel and filesystem to execute)

The Linux kernel should uncompress and mount a ROM filesystem. If there are any errors, reset the board and repeat this step. Occasionally the kernel aborts on a bootup due to a corrupted flash file system. Follow steps in the MicroMonitor tips section to issue a command (uMON> tfs clean) to defrag the flash file system.

The Linux startup script is set with a Tiny Filesystem attribute that makes `'startlinux'` an executable like a batch file in DOS. To change this attribute so that you can automatically boot with query into Linux with a query to stop after MicroMonitor boots up, issue the following command after boot up of MicroMonitor:

```
uMON> tfs -feB cp startlinux startlinux  (cr)
```

WARNING: `'-feB'` is case sensitive and if you use a lower case `'b'` in the switch option, you will not be able to boot into uMON again without re-flashing through the JTAG, which is complicated and time consuming. To change the attribute back to manual boot, substitute `'-fe'` in place of `'-feB'` in the above command above.

Step 7. Run Example Applications

When the GX-Linux boots to a Linux command shell prompt, you can enter the following commands in the serial terminal to get acquainted with all of the example programs loaded in the `'/usr/bin'`; each example file has a `'.x'` file extension. In the serial terminal command line interface window, issue the following commands:

```
# cd /usr/bin      (cr)
# ls               (cr)  (shows all examples; to execute examples, perform the next step)
# hello.x          (cr)  (executes hello world program)
```

There are approximately 20 other programs that can be executed in the same manner, and each filename ends in `'.x'` extension. Tryout as many of the examples as you wish. Read the next section to become more familiarized with MicroMonitor.

Step 8. How to Add Your Own Applications

The fastest way to get your own application on the target board is to copy your application – named with a '.x' file extension -- into the examples directory, which is under the build directory, and issue the following command in the xterm/Bash shell:

```
$ make install-apps romfs (cr)
$ make load               (cr)
```
(reload your images to the target board)

After uploading three files to the target board (startlinux, zImage, and romfs.img), you can manually start GX-Linux by issuing the following commands:

```
$ (cr)                  (get a command prompt after upload)
$ reset (cr)            (get the board in a refresh state)
$ startlinux (cr)       (Linux boots to a user prompt '#')
```

On the target board in the '/usr/bin' directory are a dozen example programs that can be run from the Minicom interface. Issue the following commands:

```
#  cd /usr/bin (cr)
#  ls (cr)
```

You can see all examples; they have a '.x' file extension on them. To run each example, issue the by-name command:

```
#  double.x (cr)
#  float.x  (cr)
#  hello.x  (cr)
etc…
```

Step 9. Useful Linux Commands

```
# cat /proc/cpuinfo
```
Show information such as bogomips, architecture, and manufacture information.

```
# reboot
```
Reboots system into uMON.

3.3 Getting Started with Application Setup on Windows Host

This section explains how to build and load applications into the target Cogent development board using the Windows/Cygwin host provided by Microcross on the Standard and Professional Platform CD-ROMs. The Application Development Platform on Windows is a complete replica of the Standard Platform on Linux host. Since kernel and library building are not provided at this time on a Windows/Cygwin host due to several environment issues, Microcross instead chose to pre-build the kernel and libraries on a Linux host and install them on the Windows/Cygwin host for those who want a familiar Windows environment to perform their application development.

Board Connection and Initial GX-Linux Bootup
Make physical connections per the diagram layout in Figure 3.6. If connecting to a hub on a network, disregard connecting an Ethernet cross-over cable. To get the target board booted up and communicating with your host Windows computer, you will need to configure your serial port and HyperTerminal (or other serial communications program) before loading GX-Linux binaries.

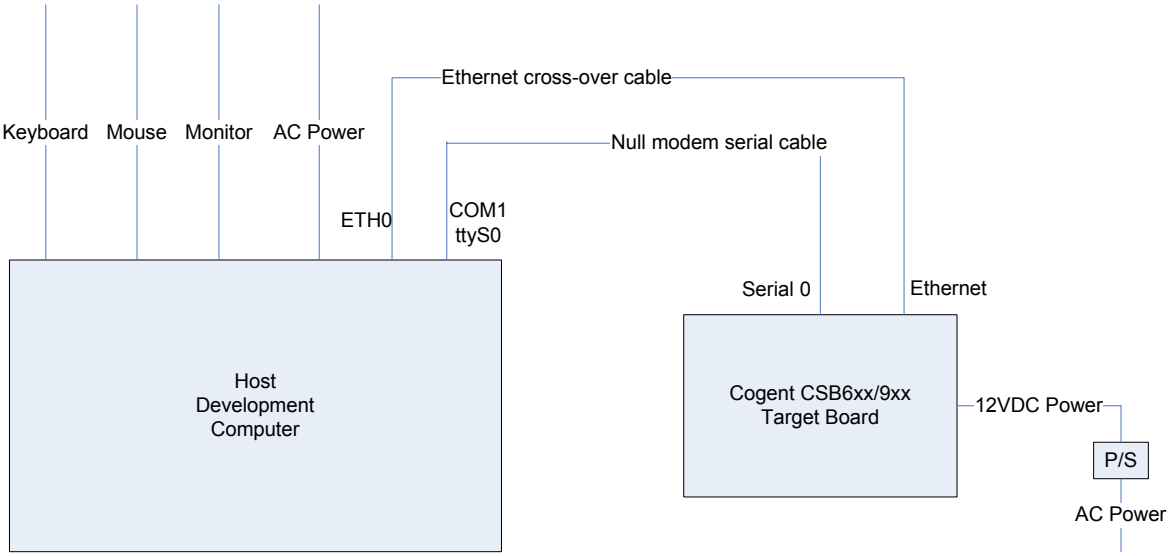
Step 1. Setup Serial Terminal

- Connect the provided null-modem serial cable to the target board and host computer's DB-9 Serial Port.
- Configure your serial terminal program to connect to the target board using the following settings:

```
38400 baud
8 data bits
No parity
1 stop bit
no flow control
```

- Connect power supply to the board, and the serial terminal should show the MicroMonitor startup banner and the uMON> command prompt.

Figure 3.6 Diagram of Board Connections to Host Computer



Step 2. Setup uMON

You have two options to setup the network file, `'monrc'` in MicroMonitor: manual settings or DHCP/BootP. Consult your system administrator if you need help in determining your network settings.

Manual network settings: The settings shown are *notional* and are for reference of the procedure only. Consult your network administrator if you do not understand how to setup network settings for your network. Use the uMON command line interface in your serial terminal session to enter these commands with your substituted settings.

```
uMON> set -c
uMON> set IPADD 192.168.0.110    (IP address)
uMON> set NETMASK 255.255.255.0 (network mask)
uMON> set GIPADD 192.168.0.250   (gateway IP address)
uMON> set -f monrc
uMON> reset
```

DHCP/BOOTP network settings: Follow this example and enter the commands using your serial terminal uMON command line interface.

```
uMON> set -c
uMON> set IPADD DHCP (or DHCPv or DHCPV or BOOTP)
uMON> set -f monrc
uMON> reset
```

Select only one for IPADD setting

`DHCP`	No verbosity enabled
`DHCPv`	Limited verbosity enabled
`DHCPV`	Extreme verbosity enabled
`BOOTP`	Runs BootP

IMPORTANT: After configuring the network settings and initiating the reset command, the target board will re-boot MicroMonitor. Notice that in the serial terminal interface that the new IP address is displayed; if the new IP address does not show up on the display, go back to the previous make the necessary corrections to `'monrc'`. IMPORTANT: When using DHCP/BootP, the new IP address should be annotated in your notes, because you will need this IP address for loading GX-Linux files via TFTP. If your DHCP/BootP server assigns a different IP address every time you reboot MicroMonitor, you may want to consider manually setting the network IP address while in the development phase of your project, because the IP address is a key parameter to your TFTP communications between the host development platform and the target board.

Step 3. Modify Environment Variables

Open Visual X-Tools or another editor and edit the master `'Makefile'` in the *build* directory. The build directory is `'Cygwin\home\gxlinux'` directory. Edit configuration variables as follows:

`ARCH = <arch-of-target>`	(i.e., arm, mips, ppc, etc.)
`BOARD = <csbxxx>`	(if not set, enter the Cogent uSBC name – e.g., csb337, csb637, etc.)
`TGTIP = <IP-address>`	(enter the IP address assigned to the target board in the last step)

Optionally, you can modify the other configuration variables that apply, but the three most important are the ones above.

Step 4. Build All

Start a GNU X-Tools Shell by clicking on the desktop icon installed. If there is no icon, you may not have the Application Development Platform on Windows installed correctly – go back to Section 2 and follow the installation instructions.

Make the example programs and create a ROM image for the `'romfs.img'` filesystem by issuing the following commands:

```
$ cd /home/gxlinux (cr)
$ make all    (cr)
```
(builds example applications, installs them, and re-generates a ROM filesystem)

Step 5. Load Binaries and Startup Script

This step assumes that the target board is powered up, connected to Ethernet, and uMON prompt is displayed in a serial terminal. From the GNU X-Tools Shell and assuming you are still in the *build* directory, issue the following command that will load the GX-Linux binaries and startup script (we also assume you assigned the target board an IP address and edited the makefile to add this IP address).

```
$ make load        (cr)
```

Three images should start loading onto the board – see the serial terminal interface for upload response in MicroMonitor. The images loaded are as follows: 1) `'zImage'` (kernel); 2) `'romfs.img'` (ROM root filesystem); and 3) `'startlinux'` (startup script).

Step 6. Reset and Boot Linux

Press the board reset switch once the files have completely transferred. From the serial terminal window, issue the following command:

`uMON> startlinux (cr)` (this script calls the Linux kernel and filesystem to execute)

The Linux kernel should uncompress and mount a ROM filesystem. If there are any errors, reset the board and repeat this step. Occasionally the kernel aborts on a bootup due to a corrupted flash file system. Follow steps in the MicroMonitor tips section to issue a command (uMON> tfs clean) to defrag the flash file system.

The Linux startup script is set with a Tiny Filesystem attribute that makes `'startlinux'` an executable like a batch file in DOS. To change this attribute so that you can automatically boot with query into Linux with a query to stop after MicroMonitor boots up, issue the following command after boot up of MicroMonitor:

```
uMON> tfs –feB cp startlinux startlinux  (cr)
```

WARNING: `'-feB'` is case sensitive and if you use a lower case `'b'` in the switch option, you will not be able to boot into uMON again without re-flashing through the JTAG, which is complicated and time consuming. To change the attribute back to manual boot, substitute `'-fe'` in place of `'-feB'` in the above command above. To stop the automatic boot, press any key while uMON is booting and before Linux boots. The bootup will stop at the uMON command line.

Step 7. Run Example Applications

When the GX-Linux boots to a Linux command shell prompt, you can enter the following commands in the serial terminal to get acquainted with all of the example programs loaded in the '/usr/bin'; each example file has a '.x' file extension. In the serial terminal command line interface window, issue the following commands:

```
# cd /usr/bin      (cr)
# ls               (cr)  (shows all examples; to execute examples, perform the next step)
# hello.x          (cr)  (executes hello world program)
```

There are approximately 20 other programs that can be executed in the same manner, and each filename ends in '.x' extension. Tryout as many of the examples as you wish. Read the next section to become more familiarized with MicroMonitor.

Step 8. How to Add Your Own Applications

The fastest way to get your own application on the target board is to copy your application – named with a '.x' file extension -- into the examples directory, which is under the build directory, and issue the following command in the xterm/Bash shell:

```
$ make install-apps romfs (cr)
$ make load               (cr)  (reload your images to the target board)
```

After uploading three files to the target board (startlinux, zImage, and romfs.img), you can manually start GX-Linux by issuing the following commands:

```
$ (cr)                  (get a command prompt after upload)
$ reset (cr)            (get the board in a refresh state)
$ startlinux (cr)       (Linux boots to a user prompt '#')
```

On the target board in the '/usr/bin' directory are a dozen example programs that can be run from the Minicom interface. Issue the following commands:

```
#  cd /usr/bin (cr)
#  ls (cr)
```

You can see all examples; they have a '.x' file extension on them. To run each example, issue the by-name command:

```
#  double.x (cr)
#  float.x  (cr)
#  hello.x  (cr)
etc…
```

Step 9. Useful Linux Commands

```
# cat /proc/cpuinfo
```
Show information such as bogomips, architecture, and manufacture information.

```
# reboot
```
Reboots system into uMON.

3.4 MicroMonitor Boot ROM Familiarity and Tips

MicroMonitor is the boot ROM that comes with each Cogent development board. It is very useful in initializing hardware, starting serial and Ethernet services, and creating a tiny filesystem to store the embedded Linux startup binaries. Moreover, MicroMonitor facilitates loading the GX-Linux binaries using TFTP services or XModem through the serial port. We will focus on the TFTP (across Ethernet) transfer solution due to the size of the images and time it takes to load binaries.

3.4.1 Manual TFTP Up-Load Procedure

Microcross ships each board pre-configured with the Linux script, `'startlinux'`, and two binaries: `'zImage'` (kernel) and `'romfs.img'` (root filesystem and Busybox utilities). To up-load your newly created versions, follow the steps in this section. The up-load address, `<target-ftp-address>`, is that of the target board.

Reload 'startlinux' Startup Script

To reload `'startlinux'` script on the target board, you have three options to control your linux bootup:

1. Load `'startlinux'` for manual bootup (requires serial connection to use the uMON command line interface to execute `'startlinux'`):

   ```
   $ cd <build-dir>/boot
   $ ttftp <target-ftp-address> put startlinux startlinux,e
   ```

2. Load `'startlinux'` for autoboot with abort query:

   ```
   $ cd <build-dir>/boot
   $ ttftp <ftp-address> put startlinux startlinux,eB
   ```

3. Load `'startlinux'` for autoboot without abort query – **warning** – there is no way to recover an error without disaster recovery efforts using a JTAG emulator to reload MicroMonitor into RAM and then write back to flash.

   ```
   $ cd <build-dir>/boot
   $ ttftp <target-ftp-address> put startlinux startlinux,eb
   ```

Change `'startlinux'` Attribute Options

There is another option to change `'startlinux'` script flags without having to reload `'startlinux'` again by using the serial uMON command line interface. Microcross', `'startlinux'`, is preloaded with the `'e'` flag attribute set, which is the manual boot up setting. Two other boot options are available and can be set issuing one of the following commands:

1. Autoboot with abort query (case sensitive).

   ```
   uMON> tfs -feB cp startlinux startlinux
   ```

2. Autoboot without abort query (case sensitive) – **WARNING** – there is no way to recover from an error without disaster recovery efforts using a JTAG emulator to reload MicroMonitor into RAM and then write back to flash.

   ```
   uMON> tfs -feb cp startlinux startlinux
   ```

MicroMonitor training material is available by Microcross. Here is a link to more information / documentation: http://www.microcross.com/html/micromonitor.html. Also, there is a MicroMonitor User Manual in the `'Cygwin/doc'` directory and the CD-ROM `'doc'` directory.

Reload 'zImage' Procedure

Similarly to the upload procedures above, you can add the `'zImage'` – Linux kernel – to the TFS on uMON.

```
$ cd <directory-linux-zImage-binary>
$ ttftp <target-ftp-address> put zImage zImage
```

Reload 'romfs.img' Procedure

Similarly to the upload procedures above, you can add the `'romfs.img'` – root filesystem and Busybox utilities – to the TFS on uMON.

```
$ cd <build-directory>
$ ttftp <ftp-address> put romfs.img romfs.img
```

To rebuild any component of this BSP, refer to Section 5 for details.

3.4.2 Other Useful MicroMonitor Commands

`uMON> tfs ls`	List files in the Tiny File System
`uMON> tfs clean`	Defrag the Tiny File System without deleting files
`uMON> tfs init`	Delete all files and defrags the Tiny File System – requires network setup
`uMON> tfs rm <file-name>`	Delete a specific file named <file-name>
`uMON> help`	Shows all of the available commands
`uMON> help <command-name>`	Shows specific help on command name
`uMON> set`	Shows all settings in memory
`uMON> flash info`	Shows all flash sectors and which ones are locked and erased
`uMON> flash unlock x-y`	Unlocks sectors x (lower bound) to y (upper bound)
`uMON> flash erase x-y`	Erase sectors x (lower bound) to y (upper bound)

Section 4.0 Visual X-Tools Example Application Project

The following steps describe the operation of the Visual X-Tools with an example Embedded Linux program for the Cogent CSBxxx. The example shown can be replicated for any of the other examples described below. The example program was created with the Visual X-Tools wizard and is available only with the professional platform or a separate purchase of the Visual X-Tools IDE for either Windows or Linux host operating systems.

4.1 Hello Example Program

This section briefly goes over the standard hello world program that talks to the console via libc-xflat.so using printf statements.

Step 1.

Startup Visual X-Tools IDE from a desktop icon or Bash/xterm shell. See the Visual X-Tools IDE icon in Figure 4.1.

Figure 4.1 Visual X-Tools IDE Icon

Step 2.

Click on `'Project|Open Workspace…'` from the menu. Navigate to the following directory:

`../<build-directory>/examples/vxtools-hello`

Open the workspace file `'hello.vxw'` and go onto the next step.

Step 3.

On the Visual X-Tools menu click on `'Build|Set Active Configuration…'` and select either `'Debug'` or `'Release'`.

Step 4.

On the Mini Toolbar (see Figure 4.2), click on `'Clean & Build Project'` icon.

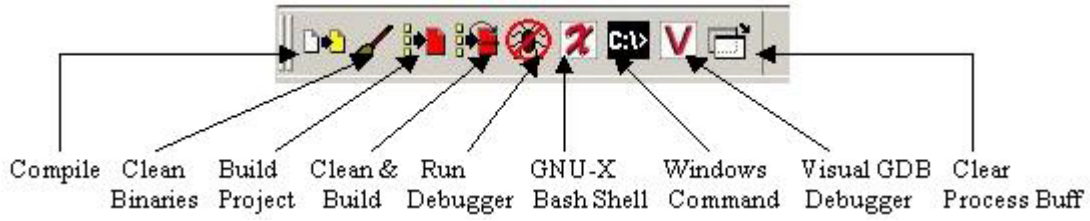


Figure 4.2 Mini-Toolbar Description

These commands are also located under the 'Build' menu. The Build window at the bottom of Visual X-Tools shows the verbose output of the build. After completion of the build, go to the next step.

Step 5.

If you selected the 'Debug' Active Configuration in Step 3, then click on 'Build|Copy Debug Image to Examples Dir'.

If you selected the 'Release' option for Active Configuration in Step 3, then click on 'Build|Copy and Strip Release to a Binary Image', then click on 'Build|Copy Release Image to Examples Dir'.

Step 6.

Issue the following commands in xterm/Bash shell for rebuilding the 'romfs.img' filesystem and replacing the existing version on root filesystem.

```
$ make install-apps romfs  (cr)
$ make load                (cr)
```

To get more specific information on how to use Visual X-Tools, consult the Visual X-Tools User Guide.

--- END OF EXAMPLE PROGRAM ---

Section 5.0 Making Modifications to GX-Linux

Every developer will have to make at least one or more modifications to GX-Linux, and depending on which Platform you are working with, the steps may vary. The following list names the appropriate section for modifying GX-Linux on your platform version.

Section 5.1
Modifying GX-Linux Standard Platform

Section 5.2
Modifying GX-Linux Professional Platform

5.1 Modifying GX-Linux Standard Platform

No changes to the Kernel and library binaries can be made using the GX-Linux scripts and makefile for the Standard Platform. Microcross ships the binary versions of the Linux kernel and libraries to make the evaluation of GX-Linux possible without incurring a major investment. The actual sources for the Libraries and Linux Kernel along with all the other sources are shipped on the GX-Linux CD-ROM per GPL requirements; however, it is up to the user to know how to build the standard products without GX-Linux help, hence the value-add of the Professional Platform. Building applications, however, can be done using the procedure in the next section. For those seeking an install and build-all out-of-the-box solution, the GX-Linux Professional Platform is recommended.

5.1.1 Building Your Own Applications

We assume you already performed at least a preliminary build of the kernel, library and root filesystem with BusyBox. You can build your own applications, load them into `'rootfs/usr/bin'` (or any other directory under `'rootfs'`) and load and run them on the target uSBC:

1. Build your application(s) and name the binary executable with a `'.x'` file extension (recommended).
2. Copy your application image file into the `'<build-directory>/examples'` directory.
3. To install your application into `'rootfs'`, re-build the ROM filesystem, and reload the target board via TFTP (assuming the target board is powered on and connected to your network), issue the following commands:

```
$ cd <build-directory>            (cr)
$ make install-apps romfs         (cr)
$ make load                       (cr)
```

If you have any load errors, make sure your target IP address (TGTIP variable) is correctly assigned in the Makefile (build directory).

5.2 Modifying GX-Linux Professional Platform

5.2.1 GX-Linux BSP Components

The software downloaded to the target hardware is comprised of various components, listed below:

- Bootloader (uMON)
- Linux kernel (Linux)
- Target file system

The target file system itself consists of various components:

- Linux file system framework including uClibc libraries
- Development applications (Busybox, tinylogin, tftp, etc)
- Loadable kernel modules
- Other applications

5.2.2 Building All of the Components

All of the GX-Linux BSP components can be built using the following commands:

```
$ cd <build-directory>  (cr)
$ make all              (cr)
```

Makes zImage (~1.2Mb), romfs.img (~1.7Mb), and startlinux (~1.3 kb) – size will vary as apps apply.

5.2.3 Building All Components Excluding Examples

All of the examples can be built using the following commands:

```
$ cd <build-directory>              (cr)
$ make distclean                    (cr)
$ make config system
```

The above step builds everything. To add your own example applications after building the system, follow the procedure below, *Building Your Own Applications*.

5.2.4 Building Your Own Applications

We assume you already performed at least a preliminary build of the kernel, library and root filesystem with BusyBox. You can build your own applications, load them into `'/usr/bin'` (or any other directory) and run them on the target uSBC:

- Build your application(s) and name the binary executable with a `'.x'` file extension (recommended).
- Copy your application files into the `'<build-directory>/examples'` directory.

- To install the your own applications, re-build the ROM filesystem, and reload the target board via TFTP (assuming the target board is powered on and connected to your network), issue the following commands:

```
$ cd <build-directory>            (cr)
$ make install-apps romfs         (cr)
$ make load                       (cr)
```

If you have any load errors, make sure your target IP address is correct in the Makefile (build directory).

5.3 Changing the Kernel Configuration

5.3.1 Configuration Files

The Linux kernel supplied with your BSP is pre-configured and ready for use. If your project requires components that are not contained in the default GX-Linux kernel, you may want to change the kernel configuration. The kernel configuration is expressed in two files: (1) the `'.config'` *working copy* and (2) `'kernel-conf'` *master backup copy*. The working copy is retained at `'.../gxlinux/linux/.config'`; the master backup copy at: `'.../gxlinux/config/<csbxxx>/kernel-conf'`.

The configuration file is read and modified by xconfig, described below. After running xconfig, another derived file, `'.../gxlinux/linux/include/linux/autoconf.h'` is created. The autoconf.h file is included by most kernel source files to control which features are built.

5.3.2 Changing the Working Configuration

Changing the GX-Linux kernel configuration requires changing the .config working copy. It should not be modified directly, but rather it should modify it using the steps below. In this procedure, we use `'make ARCH=<arch> xconfig'` to modify .config. Note: We assume that you have the QT components from the KDE development package installed before using xconfig.

1. Type the commands below to initiate xconfig:

```
$ cd <build-directory>/linux      (cr)
$ make ARCH=<arch> menuconfig     (cr)
```

   Or alternatively:

```
$ make ARCH=<arch> xconfig        (cr)
```

   Note: ARCH can equal `'arm'`, `'mips'`, `'ppc'`, etc.

2. The menuconfig/xconfig will display all of the kernel options and their current settings. Menuconfig/xconfig will allow you to change any kernel setting. Be sure to save your changes before you exit from xconfig.

```
$ cd <build-directory>  (cr)

$ make SaveCfg          (cr)
```

   (saves the `'.config'` file to the master config file in `'config/csbxxx/kernel-conf'` directory)

3. With the menuconfig/xconfig session complete, you can build the new GX-Linux kernel as follows (assuming arm for purposes of this example):

```
$ cd <build-directory>/linux                    (cr)
```

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- mrproper    (cr)
$ make ARCH=arm CROSS_COMPILE=arm-linux- vmlinux     (cr)
```

If the build is successful, the resulting kernel image can be found in the Linux directory:

<*buld-directory*>/linux/vmlinux -- an ELF object format

Now you can use the master configuration and build system with confidence that the kernel will build.

5.3.3 Resetting the Kernel to the Default Configuration

Should you want to return to the GX-Linux default settings, you can accomplish this with the steps below (assuming the ARM in this example):

```
$ cd <build-directory>                                          (cr)
$ make distclean                                                (cr)
$ cd linux                                                      (cr)
$ make ARCH=arm CROSS_COMPILE=arm-linux- mrproper               (cr)
$ cp arch/<arm-or-other>/configs/<csbxxx_defconfig> .config (cr)
$ make ARCH=arm CROSS_COMPILE=arm-linux- oldconfig              (cr)
$ make ARCH=arm CROSS_COMPILE=arm-linux- vmlinux                (cr)
$ cd ..                                                         (cr)
$ make SaveCfg                                                  (cr)
```

5.3.4 Customizing the Kernel Configuration

If you find that your particular needs cannot be met using the available kernel configuration options, you may want to modify the 'make xconfig' scripts that control the configuration session. The particular file that controls the xconfig program presented above is at:

…/gxlinux/linux/arch/arm/config.in

5.4 Changing the Target File System

The directory structure under '…/gxlinux/rootfs' is the master file system structure used to create the target file system. Files can be installed into the target file system by simply copying them into the appropriate sub-directories under '…/gxlinux/rootfs'. The target file system is built by default as a compressed 'romfs.img' file system using genromfs.

5.4.1 Adding Content to the Target File System

You can configure the target file system with additional content not included in the BSP's default target file system. The following steps illustrate two methods that can be used to put new applications into the target file system. In these illustrations, the application is the "Hello, World" example.

1. The following steps compile and install the "Hello, World" example into the target file system image on the Linux development workstation.

```
$ cd <build-directory>/examples/hello (cr)
$ make apps install-apps romfs        (cr)
```

The make 'romfs.img' target command creates the target file system image as '<build-directory>/romfs.img', which is the file we downloaded to the target hardware using the uMON bootloader.

5.5 Putting the New System on the Target Hardware

5.5.1 Putting the New System on the Target Hardware

Once you have created a new BSP component (a new kernel image, a new file system image, and the uMON 'startlinux' script), the uMON bootloader can be used (a) to download the components into the target hardware's SDRAM, (b) to save the component in FLASH memory, and/or (c) to boot the new system. The steps below describe the general procedure to accomplish this. See the uMON documentation or Section 3 of this User Guide for more detailed instructions.

1. Power cycle the target hardware. Start your serial terminal that is connected to the target board, and if configured correctly, you should get the uMON prompt.

2. From xterm/Bash shell, change to the *build* directory and issue 'make load' to download the new kernel, file system, and startlinux script to the TFS.

3. Issue the 'startlinux' script on the uMON command line in the serial terminal.

   The 'startlinux' script is a uMON script that executes the required uMON commands to boot the Linux kernel with the right configuration. The configuration values in 'startlinux' were *auto-generated* by the GX-Linux BSP Configuration Tools according to the values selected using the BSP configuration tool.

   The Linux kernel and file system image must have been saved to TFS prior to executing the 'startlinux' script. Further, the 'startlinux' script will require that the Linux kernel and file system image have the same names in TFS as they did on the Linux workstation. The 'startlinux' script should be stored on TFS with the "executable script" attribute enabled (the 'make load' command automatically handles this).

5.6 Read-Only and Read-Write File Systems

5.6.1 ROMFS File System

The default file system used in the GX-Linux BSP is not a permanently writable file system. Anything written to the romfs will be lost when the target hardware is powered down. Compact Flash, SD/MMC, USB Pendrives and other permanent memory storage devices can be used, but GX-Linux is not pre-configured for any other memory devices. You can, however, create an NFS mounted root filesystem on your network and simply access it using the 'startlinux' script. You can also create a filesystem on a USB pendrive and mount it automatically at startup.

5.6.2 NFS Mounted Root Filesystem

To mount an NFS read-write root filesystem, you need to have a server setup for NFS mounting. The procedures to perform this task are beyond the scope of this User Guide and should be done by a competent Linux system administrator. The administrator needs to create the NFS server in a location on your network that you can place your 'rootfs' directory on and have the correct dev directory. When you 'make all', two sets of devs get created: 1) initial set of 'pseudodevs' for 'romfs.img'; and 2) final set of 'devs' for 'rootfs' to become a NFS mountable root filesystem. The pseudodevs are all prefixed with '@' symbol, while the real devs are not. Two pieces of information are important to use GX-Linux with an NFS mounted filesystem: 1) the NFS server IP address; and 2) the NFS 'roofs' path on the server. Recall that 'rootfs' is part of the build directory in

'gxlinux'. You can copy this directory after 'make all' and put it on another machine for mounting to, or you can simply make your host system an NFS server and add the path to 'rootfs' in the GX-Linux build directory.

Once the target filesystem is mounted via NFS, then applications can be written or modified under this filesystem, which is visible after the next power cycle. In this scenario, the target filesystem is not hosted in the target hardware but rather on a hard drive of a remote Ethernet connected machine, typically your Linux development workstation.

How to Mount Using an NFS Server after Bootup Using the romfs.img

Issue the following command:

```
# mount –t nfs –o nolock <ip-address>:/<dir-to-mount> /mnt/nfs (cr)
```

Example

```
# mount –t nfs –o nolock 192.168.0.2:/home/james/gxlinux/rootfs /mnt/nfs (cr)
# cd /mnt/nfs (cr)
# ls (cr)
```

How to Start GX-Linux with NFS Mounted Automatically or Manually from the Startlinux Script

The developer has a number of options as to how to automate the mount of an NFS server, but the most common way with GX-Linux is to setup environment variables in uMON to point to the server IP and path, and then issue the 'startlinux' command or build an automated script and set it with autoboot with query attributes.

1. Add two 'monrc' Environment Variables: NFSSERVER and NFSPATH
   - uMON> set NFSSERVER <IP-ADDRESS> (cr)
   - uMON> set NFSPATH /path/to/rootfs (cr)
   - uMON> set –f monrc (cr)
   - uMON> reset (cr)
2. Manually Start Linux from Serial Terminal
   - uMON> startlinux nfs $NFSSERVER $NFSPATH (cr)
3. Autostart Linux from uMON Bootup
   - uMON> edit –feB startlinuxnfs (cr)
   - uMON> i (cr)
   - uMON> startlinux nfs $NFSSERVER $NFSPATH (cr)
   - uMON> . (cr)
   - uMON> q (cr)
   - uMON> reset (cr) (autostarts with query into linux with NFS mount)
   - # reboot (reboot from Linux to query uMON – hit any key to stop Linux boot)
4. Modify Autostart File Attributes to Manual Start Attribute with TFS Copy Command
   - uMON> tfs –fe cp startlinuxnfs startlinuxnfs (cr) (the –fe gives the file an attribute for executing a script)
   - uMON> reset (cr)
   - uMON> startlinuxnfs (cr) (start Linux nfs mode manually)

The procedure in 3 and 4 above can be done also for a 'startlinux' without NFS; simply omit 'nfs $NFSSERVER $NFSPATH' from the command line in the Autostart procedure. If there are any errors, consult your system administrator for help, or manually modify 'startlinux' in the '…\config\$(BOARD)\' directory on the host workstation to hard-code your NFS settings (NFSSERVER and NFSPATH); reload the script into the uMON TFS (tiny file system), and then issue the command in step 2 above.

5.6.3 Writing to On-Board Flash via a Pendrive (USB)

Some Linux applications will require the ability to store data persistently in flash. Such applications will require extensions to the target hardware to allow hosting of a read/write flash file system. These extensions will include:

- Provision of a writable flash file system that the kernel can mount on bootup. Writable flash file systems are available under the 2.6 kernel (ext2, ext3, JFFS, JFFS2, VFAT, etc.). These flash file systems are well documented and discussed in the Linux Open Source embedded systems community.
- Kernel configuration changes (for example, via make xconfig or menuconfig) to enable the particular file system type used.
- Addition of an underlying flash media access layer/driver to support the board's flash chip(s) or external compact flash pendrive or USB drive.

To mount a pendrive using the standard configuration of GX-Linux and USB host. Insert the pen drive and the operating system will echo the mount point; issue the following command:

```
      # mount  -t vfat /dev/sda1  /mnt/pendrive (cr)
or alternatively:
      # mknod /var/sda1 b 8 1
      # mount –t vfat /var/sda1 /mnt/pendrive

for a standard 'ext' filesystem, issue the following command:
      #mount /dev/sda1/  /mnt/pendrive  (cr)
or alternatively:
      # mknod /var/sda1 b 8 1
      # mount /var/sda1 /mnt/pendrive
```

After mounting the pen drive, you can change to this drive or copy files to it using the mount point `'/mnt/pendrive'`. Use the `'umount /mnt/pendrive'` command to un-mount the drive.

5.6.4 How to Prepare a Pendrive with a Root Filesystem and Work with GX-Linux

Start an xterm or Bash shell and log in as root or `'su'`. We assume that `'sda1'` is the USB device in this example. Insert the pendrive into your host computer that has GX-Linux installed. Mount the pendrive.

Example

`# mount /dev/sda1 /mnt/pendrive (cr)` (create the pendrive directory in `'/mnt'` if it doesn't exist)

`# df (cr)` (reports back the number of blocks for all physical devices)

Note the number of blocks reported for `'sda1'` or your USB device name.

```
# umount /mnt/pendrive  (cr)
# /sbin/mkfs /dev/sda1 <#blocks>  (cr)
```

Ask your system administrator how to format filesystems if you are not sure how to use this command.

```
# mount /dev/sda1 /mnt/pendrive  (cr)
# cd /mnt/pendrive  (cr)
# cp –R /<path-to-rootfs>/gxlinux/rootfs/*  .  (cr)
# ls dev (cr)  (if this directory is empty, then you do not have any device nodes, read below)
```

Make sure `'rootfs/dev'` directory is populated with device nodes, because they get deleted when a `'distclean'` is run. If `'dev'` directory is empty, then issue a `'make all'` in GX-Linux before issuing the above command.

```
      # umount /mnt/pendrive  (cr)
```

Now pull the pendrive out the host machine and with power off the target board, insert the USB pendrive into the USB host socket and power on the target board (assuming serial and Ethernet are connected). At the MicroMonitor umon prompt, issue the 'startlinux usb' command (assuming you have GX-Linux files already loaded). The pendrive should be mounted as your root filesystem; if not, then troubleshoot the pendrive on your host computer to see if the root filesystem looks like the same one in your rootfs directory. Follow the procedures in previous sections to get NFS working to validate the root filesystem.

5.6.5 Writing to On-Board Flash via MMC/SD Card

Check to see if 'mmc' is available in the device driver 'devices' list:

```
# cat /proc/devices (cr)
```

If 'mmc' is listed on the Block devices, then you may issue the following command sequence manually or add to the '/etc/init.d/rcS' file in rootfs and varetc to be executed automatically at startup. The 'rcS' file is like an 'autoexec.bat' file in DOS that executes commands as given at startup. The 'microcross-banner' is executed in this manner.

Th 'mmc' block device will have a number associated with it when the above procedure is executed, and the major number is used in making a node.

Example:

```
# mknod /var/mmcblk0p1 b 254 1 (cr)

# mount –t vfat /var/mmcblk0p1 /mnt/mmc (cr)
```

Leave the '-t' off for ext2 and ext3 filesystems.

For some Cogent boards, the 'startlinux' script has the feature to mount a root filesystem that has been created on an SD or MMC card. To automount this filesytem, issue a 'startlinux mmc' command in uMON. Optionally, you can create another startlinux script, say 'startlinuxmmc', and put on line of code in it to call 'startlinux' with the 'mmc' argument. When TFTPing the new script, add the 'eB' attribute to autostart Linux with query from uMON.

5.6.6 Writing to On-Board Flash via a Compact Flash Card

The Compact Flash (CF) socket on Cogent boards is configured as a mass storage device (IDE) by default with GX-Linux. Contact Microcross for information on other options available (e.g., PCMCIA, WIFI, Wirelss Ethernet, etc.).

The compact flash card driver can be mounted with the following command:

```
# mount /dev/hda1 /mnt/cf (cr)
```

Note: We have noted in our testing that CF cards 512 MB and larger do not work in the Cogent boards. Recommend using 256 MB or smaller memory cards.

Section 6.0 Debugging Applications

There are two common tools used for symbolic debugging software used in Linux based target hardware. The GDB debugger is typically used to debug applications. A JTAG emulator is typically used to debug the Linux kernel and kernel device drivers. Debugging using a JTAG emulator is dependent on the emulator used, and thus is not discussed in this manual.

6.1 The GDB Debugger

6.1.1 GDB Debugger

The purpose of a debugger is to allow the developer to see what is going on inside his/her program while it executes. Two debuggers are provided in the GX-Linux BSP, and they are located in `'/usr/bin'`; the toolnames are `'<target-alias>-gdb'` and `'<target-alias-gdbtk'` (e.g., arm-linux-gdb and arm-linux-gdbtk). GDB is the command line debugger, and GDBTK is the Visual GDB (aka Insight). Use of that debugger is the topic of this section. For detailed information regarding usage of the GDB debugger, please refer to one of the following:

- The info and standard GDB man pages included on your Linux workstation.
- The online manual provided by the Free Software Foundation at http://www.gnu.org/software/gdb/documentation.
- Debugger docs on the GX-Linux CD-ROM under docs directory

6.1.2 gdbserver

The GDB debugger executes on the user's Linux workstation. In order to communicate with a program running on the target hardware, an additional program must run on the target hardware. This program is gdbserver. It is included with the GX-Linux toolchain and is installed in the target file system as `'/usr/bin/gdbserver'`.

6.1.3 gdbserver Communications

There are two methods for communication between the target hardware's gdbserver and the Linux workstation GDB client. One method uses an available serial port while the other uses the Ethernet port.

Some target hardware have multiple serial connectors with one of them dedicated to the system console (typically `/dev/ttyS0`) and another (`/dev/ttyS1`) is available for GDB communication with the Linux workstation. In the case where the target hardware has only one serial port, it is probably not available for GDB debugging. In this case, the Ethernet port should be used for GDB communication with the Linux workstation.

If you are going to use the serial port for debugging, see the 6.4 Changing Serial Port Settings section for additional information about the use and configuration of the board's serial port. Use the connection that is appropriate for your target hardware, specifying it at the time you start the debug session.

6.1.4 Graphical Front-Ends

If you would like to use a graphical front end with arm-linux-gdb, there are several compatible front ends available. One popular front end is the ddd utility (http://www.gnu.org/software/ddd). Below is an example of how you would invoke the ARM9 arm-linux-gdb debugger, using the ddd graphical front end:

```
$ ddd -debugger arm-linux-gdb (cr)
```

6.1.5 Debug-able Files

You can use the '<target-alias>-gdb' debugger (with the gdbserver) to debug Linux programs on the target hardware. You can use any program invoked from the Linux command line with this debugger, *provided that* the program was built to be compatible with GDB.

6.1.6 Visual GDB

To debug using the Microcross Visual GDB (a.k.a. Insight), refer to the Microcross documentation, which is offered online at www.microcross.com.

6.2 Debugging Examples

To quickly get started debugging, several example files were included in the GX-Linux package. A list of all the examples follows:

6.2.1 Example with Visual GDB

Step 1. Start gdbserver on the Cogent uSBC

Using the Minicom interface to Linux running on the Cogent uSBC, issue the following command:

```
# gdbserver localhost:2001 /usr/bin/div.x  (cr)
```

Step 2. Start Visual GDB

Issue the following set of commands in xterm shell (the directory information is notional – use your path setting).

```
$ cd /home/steve/gxlinux/examples   (cr)
$ arm-linux-gdbtk div.x             (cr)
```

Click on 'File|Target Settings…' and select 'GDBserver/TCP' from the drop-down list. Enter the 'Hostname' – the Microcross pre-assigned host is the IP address: 192.168.0.110. Next, enter the 'Port' number, and the Microcross port number assigned is '2001'. Click 'OK' and click on the 'Run|Connect target' menu item. Set your breakpoints and click on the 'Run' icon. You should now be able to single step through the example program.

6.2.2 Example with GDB

Debugging the program: Once you have placed your executable "ELF" program in the target hardware's file system, it is ready for debugging using the following steps. On the Linux workstation, you will need the same, debug-able "ELF" version of the program in the same directory where you will invoke arm-linux-gdb.

1. **Start the gdbserver.** On the target hardware, invoke the gdbserver in the directory where you have the file you are debugging: the "flat" or "xflat" format version of your program. To start the target hardware-side of the GDB session, do one of the following:

   If you are using a network GDB connection, enter the command below:

```
# cd /usr/bin                (cr)
# gdbserver :5472 hello.x    (cr)
```

The port number 5472 was selected randomly as a high port number. Any high port number that doesn't conflict with existing active ports will work. The command line also indicates the name of the program you intend to debug, hello in this example. This program is the debug-able "ELF" program that we built and installed above.

If using a serial GDB connection, enter the commands below. See the Changing Serial Port Settings section for instructions on building the stty command into busybox.

```
$ stty -F /dev/ttyS1 -iexten -echoctl 38400 (cr)
$ <target-alias>-gdb /dev/ttyS1 hello.x     (cr)
```

Example: In either case, gdbserver will respond with something like the following example:

```
# cd /usr/bin                          (cr)
# /usr/bin/gdbserver :5472 hello.x     (cr)
Process hello created; pid = 26
```

2. **Connect <target-alias>-gdb.** Start the Linux workstation-side of the GDB session by invoking the <target-alias>-gdb debugger. This establishes a connection to the GDB server previously started on the target hardware. We invoke <target-alias>-gdb while in the directory containing the debug-able version of our "ELF" program file, hello. Then issue commands at the GDB command line prompt to establish the connection and load the program's debug symbols.

   If you are using the `'ddd'` graphical front end, invoke it using the following command.

   ```
   $ ddd -debugger <target-alias>-gdb              (cr)
   ```

   If using a network GDB connection, enter the commands below.

   ```
   $ cd <build-directory>/examples/hello.x        (cr)
   $ <target-alias>-gdb                           (cr)
   (gdb) set endian little                        (cr)
   (gdb) target remote x.x.x.x:2001               (cr)
   (gdb) add-symbol-file hello.x                  (cr)
   ```

   Where:

   - The string *x.x.x.x* is replaced with the actual IP address of the target hardware. (If you are not sure what the IP is, type `'ifconfig'` on the target hardware command line to generate a network configuration report).
   - The port 2001 was selected randomly as a high port number. It must match the port number that was provided on the preceding target hardware gdbserver command.

   If using a serial connection to the gdbserver, enter the commands below.

   ```
   $ cd <build-directory>/examples/hello.x        (cr)
   $ <target-alias>-gdb                           (cr)
   (gdb) set endian little                        (cr)
   (gdb) set remotebaud 38400                     (cr)
   (gdb) target remote /dev/ttyS1                 (cr)
   (gdb) add-symbol-file hello.x                  (cr)
   ```

   By default, most Linux workstation's serial ports start up at 9600 baud. On the target hardware `'/dev/ttyS1'` starts up at 9600 baud. See the Changing Serial Port Settings section for

instructions on building the stty command into busybox which will allow you to change the baud rate on `'/dev/ttyS1'`.

6.3 Debugging Using the .gdbinit

Using ~/.gdbinit. The <target-alias>-gdb/gdbtk debuggers will read an optional ~/.gdbinit when they are started. For either gdbserver network or serial connection, the corresponding set of GDB commands can be defined as a macro within your ~/.gdbinit to simplify these steps.

Consider the following sample ~/.gdbinit:

```
define target_remote
  set endian little
  target remote x.x.x.x:2001
end
```

With this macro defined in your ~/.gdbinit file, GDB can be connected to the remote gdbserver with the single command:

```
(gdb) target_remote
```

7.0 Networking

7.1 Default Kernel Configuration with Networking Enabled

The GX-Linux BSP ships with a default kernel configuration that has networking enabled. With the supplied default configuration, you will only need to use the uMON to set networking values appropriate for your network. After connecting the target hardware to a LAN and booting the kernel, you should be able to use the network. The uMON passes the network settings to the Kernel via the `'startlinux'` script.

For example, commands like the ones given below should be immediately available to you.

```
# ifconfig (cr)
# route (cr)
# ping <known-network-IP-address> (cr)
# ...etc...
```

7.2 Network Configuration

The network is automatically configured to mimic the settings in uMON when GX-Linux boots; however, for modification or adding a second Ethernet configuration (for those targets with dual Ethernet), follow these simple set of instructions.

The Ethernet driver and board configuration are setup for eth0 at install time. For boards with dual Ethernet, the board is labeled ENET0 and ENTET1, where ENET0 is mapped to eth0 and ENET1 to eth1. After Linux bootup, to enable a second ethernet device, you can manually enter the following commands in the serial terminal window:

```
# ifconfig eth1 <ip-address> netmask <netmask> up
```

Example

```
# ifconfig eth1 192.168.0.111 netmask 255.255.255.0 up
```

To Disable either eth0 or eth1, do the following:

```
# ifconfig eth1 down
```

or

```
# ifconfig eth0 down
```

You can re-enable either port if no reboot has occurred by issuing:

```
# ifconfig eth1 up
```

or

```
# ifconfig eth0 up
```

Issuing `'ifconfig'` and (cr) shows the status of what is up and down with the dual Ethernet devices. Note: MAC addresses are assigned from what is set from the factory, and they are labeled on the uSBC module. If the uSBC supports two Ethernet MACs, they will be assigned one LSB difference. For example:

```
00:23:31:66:00:20  for eth0
00:23:31:66:00:21  for eth1
```

The user may add a command sequence manually or add it to the `'/etc/init.d/rcS'` file in rootfs and varetc to be executed automatically at startup. The `'rcS'` file is like an `'autoexec.bat'` file in DOS that executes commands as given at startup. Rebuild and load the romfs image to take the changes to `'rcS'`.

Example from the Build Directory:

```
# make romfs
# ttftp <target-ip> put romfs.img romfs.img
```

8.0 Serial Port

8.1 Serial Devices

8.1.1 Introduction

In this section we discuss the serial port and the two linux devices listed below.

- /dev/ttyS0
- /dev/ttyS1
- /dev/console

8.1.2 Default Serial Driver Configuration

The GX-Linux BSP ships with a default kernel configuration that has the appropriate serial driver enabled, as shown in the table below:

- /dev/ttyS0 (Used for console)
- /dev/ttyS1 (Available)
- /dev/console (Mapped to /dev/ttyS0)

8.2 Program Access to /dev/ttyS0

With serial port support enabled you can use the `'/dev/ttyS0'` Linux device to programmatically send and receive characters over the serial port. For reference, the make xconfig setting has the effect of enabling the CONFIG_AT91RM9200_SERIAL definition used during the build of the kernel. If you hardware supports `'/dev/ttyS1'`, this will enable the second serial port as well.

8.3 System Console

8.3.1 Default Console

The `'/dev/console'` is the device underlying system's stdin, stdout, and stderr file descriptors. In the GX-Linux BSP, the kernel first tries to open `'/dev/console'` when it boots to serve as the console. Failing that, it will try to open `'/dev/ttyS0'`.

8.3.2 Login via the Serial Port

If you attach a terminal or a computer running terminal emulation software to the target hardware's serial port, then you can login to the target hardware via the serial port.

8.3.3 Standard IO

Once logged in over the serial port, the system console is automatically mapped to the `'/dev/ttyS0'` device. This means that normal printf() statements in a program will route out the serial port. Or put generically, any

reference to the system's stdin, stdout, and stderr device descriptors will map to the `'/dev/ttyS0'`, and therefore the serial port.

8.3.4 Restrictions

Because `'/dev/ttyS0'` is the default console, applications should NOT use it for other purposes. In cases where the board has a second serial port, applications can instead use `'/dev/ttyS1'`.

8.4 Serial Settings

Default Baud Rate

By default, the target hardware `'/dev/ttyS0'` serial port starts up at 38400 baud and `'/dev/ttyS1'` serial port, if available, or starts up at 9600 baud (most Linux workstation's start up at 9600 baud). The `'/dev/ttyS0'` baud rate value of 38400 was chosen to be compatible with the uMON target hardware bootloader. A single terminal session (such as Minicom) can communicate with the bootloader and then with the booted kernel without changing any settings. Applications that use a different baud rate can use the standard stty utility or ioctl() calls on `'/dev/ttyS0'` or `'/dev/ttyS1'`, as appropriate, for changing serial port speed at runtime.

The default `'/dev/ttyS0'` settings are:

- 38400 baud
- 8 data bits
- no parity
- 1 stop bit

The default `'/dev/ttyS1'` settings are:

- 38400 baud
- 8 data bits
- no parity
- 1 stop bit

8.5 Kernel Console over Serial

Default Configuration

In the make xconfig, you will encounter the option "Console on Amtel AT91RM9200 serial port". This feature is enabled by default. This is what allows the internal kernel messages to appear over the serial line and therefore on your remote connected terminal session (such as Minicom). Recall that printk() is NOT the same as printf():

- printk() is only performed within the kernel itself for various diagnostics and OS warning messages.
- printf() is used by applications in user space for general console output.

The setting we use here only applies to the kernel printk() messages.

Section 9. How to Use GNU X-Tools

9.1 Introduction

This section is intended to provide an abbreviated overview of using the GNU compiler and tools to generate executable programs for a target processor. Mastering the toolsuite and using it to develop embedded systems on a practical scale requires greater in-depth knowledge than can be covered here. If you are considering the GNU cross-tools for full-scale development, it is suggested that you acquire the manuals listed in the bibliography (see Bibliography). The ones having unrestricted reproduction rights are included in the docs directory on the GNU X-Tools CD-ROM. The O'Reilly book, *Programming with GNU Software*, by Mike Loukides and Andy Oram, is highly recommended, being complementary to this User Guide as well as a comprehensive guide to the GNU tools. The remaining texts provide more in-depth knowledge for specific toolsuite components as needed. In addition, Microcross offers a *GNU X-Tools Training Guide* as a complimentary product. It is based on an on-site three-day training course for beginning to intermediate level audiences. The course material is tailored to the GNU X-Tools for embedded development, and the format is geared to a self-paced training program.

9.2 Using the GNU X-Tools Command Line Tool (`xtools`)

The GNU X-Tools command line tool is a shell tool designed to provide several functions that facilitate convenient management of the GNU X-Tools cross-toolsuites from a command shell environment. For many developers, this method is more convenient than using a GUI. The following functions may be invoked from any `'Bash'` Shell prompt:

1) `xtools (enter)` -- Execution of the `'xtools'` command without arguments will show an abbreviated help screen that summarizes the command options described in this section. In addition, execution of an incomplete command (insufficient or incorrect arguments) will produce a help dialog for that command.

2) `xtools <target-alias>` -- This command starts a new bash shell having the appropriate environment variables and aliases initialized such that the cross-toolsuite components for the designated target are invoked by the familiar command names, in lieu of the native tool chain elements. For example, after starting a shell using the command line `'xtools arm-elf'`, the command `GCC` would actually start the compiler `'arm-elf-gcc'`. This facility allows the command line user to execute the cross-tools using the same command names as used in the native shell environment. This remapping of tool command names remains in effect until the shell is exited via `'Ctrl-D'` or the `'exit'` command. The GX-Linux source tree is configured for cross-compilation, so the command is not required for any builds or makefile targets.

3) `xtools status [path prefix]` -- This command generates a summary screen which displays the installation status of each GNU X-Tools cross-toolsuite in terms of the types, counts, and locations of files associated with each toolsuite. The command displays the toolsuite status by target-alias vs. the population of files in the following format:

```
Column 0 Target Name (target-alias) for this toolsuite
Column 1 (/usr/bin/<target-alias-*) files
Column 2 (/usr/<target-alias>/bin/*) files
Column 3 (/usr/<target-alias>/lib/*) files
Column 4 (/usr/<target-alias>/include/*) files
Column 5 (/usr/lib/gcc-lib/<target-alias>/*) files
Column 6 (/usr/man/man1/<target-alias>-*) files
Column 7 Total file count for this toolsuite
```

The optional path prefix argument allows the utility to examine the installation state in which the user may have elected to manually install, or build/install toolsuite(s) into an alternate location (other than `'/usr'`).

4) `xtools install <target-alias> [archive-path]` -- This command will install a complete cross-toolsuite from the distribution tarball or zipset located on the specified archive path. All files are unarchived to the proper location (see above) regardless of the current directory state, and subsequent invocation of a bash shell using the '`xtools <target-alias>`' command will render the toolsuite ready to use. If the archive path is not provided, a default path is selected. The command '`xtools install`' will display the default path, which is '`/mnt/cdrom`' unless it has been changed or overridden by the global environment variable '`$XTOOLS_ARC_PATH`'.

5) `xtools remove <target-alias> [path prefix]` -- This command allows the user to selectively remove a designated toolsuite from the system. In the unusual case where the toolsuite has been manually installed to some root prefix other than '`/usr`', the second option can be used to specify the root prefix (such as '`/usr/bin`') from which removal of all toolsuite files is desired. This command is most useful if you have performed a build from sources and subsequent install to an alternate root prefix path, and now desire to remove the installed tools.

6) `xtools remove-all go [path prefix]` -- The '`remove-all`' command allows the user to summarily remove all toolsuites from the system. With the exception of the files '`/usr/bin/xtools`', '`/usr/bin/xtools.rc`', and '`/etc/profile`' (Cygwin only); this command affects a complete removal of GNU X-Tools toolsuite binaries from your system.

7) `xtools install-libs <target-alias> [arcpath]` -- This option allows an administrator to update or reinstall toolsuite libraries only. This feature is useful for selectively reinstalling libraries that have been over-written, or updating the libraries from a newer toolsuite version without updating the entire toolsuite. It updates the following files:

```
/usr/<target-alias>/lib/*/libc.a
/usr/<target-alias>/lib/*/libstdc++.a
```

It extracts the library files from a Microcross GNU X-Tools Toolsuite distribution CD-ROM, or filesystem image. The optional '`[arcpath]`' parameter may be used to locate the update files. **Note:** This feature is currently experimental and may change.

8) `xtools install-hdrs <target-alias> [arcpath]` – This option, like option 7, allows incremental updates to an installed toolsuite except that it updates the library headers files. It updates the following files:

```
/usr/<target-alias>/include/*/*.h
```

9.3 Invoking GNU X-Tools

To invoke GNU X-Tools from the command line in the shell, type '`xtools target-alias`' and from that point on you do not need to type the alias at any point further. The target-alias will become part of the command line prefix (i.e., '`target-alias$`'). To change or exit the target-alias mode, type '`exit`' at the command line.

From Command Shell, type: `$ xtools arm-linux`
The prompt will now show: `arm-linux$`

Issuing the following commands can be accomplished using the shortened version of tool name (i.e., `gcc`, `as`, `ar`, etc.), provided that you perform the '`xtools <target-alias>`' command as shown above. All GX-Linux sources are pre-configured for cross-compilation, so the xtools script is not necessary.

- *target-alias*-**gcc**
  Invokes all the necessary GNU compiler passes for the specific target processor toolsuite using (i.e., arm-elf, mips-elf, ppc-elf, etc.).
- *target-alias*-**cpp**
  Invokes the preprocessor that processes all of the header files and macros that the target requires.
- *target-alias*-**gcc**
  Invokes the C compiler that produces assembly language code from the processed C files.
- *target*-alias-**g++**
  Invokes the C++ compiler that produces assembly language code from the processed C++ files.
- *target*-alias-**gdb**
  Invokes the GNU Debugger with command line input (see Debugging with GDB in the PDF files on the GNU X-Tools CD.
- *target*-alias-**gdbtk**
  Invokes the GNU Debugger with a visual GUI interface called Visual GDB by Microcross (a.k.a. Insight).
- *target-alias*-**as**
  Invokes the GNU assembler that produces binary code from the assembly language code and puts it into an object file.
- *target-alias*-**ld**
  Invokes the linker that binds the code to addresses links the startup file and libraries to the object file and produces the executable binary image.

9.4 Using the GNU X-Tools Toolsuite

The first step to developing code with the GNU X-Tools toolsuite is creating and editing the source code. Microcross provides the Visual X-Tools IDE as companion product for code editing and project management; however, a programmer can use any text line editor to create source files and makefiles and use the GNU X-Tools as the production build environment. Free editors under Cygwin include VIM, an improved VI editor clone, and Xemacs, a powerful editor and project manager. Under Linux, the user also has a wide variety of choices including the Microcross Visual X-Tools IDE and GNU Xemacs edit (see docs in the Linux distribution).

9.4.1 Simple Example

Start a shell (xterm on Linux and GNU X-Tools Shell on Windows/Cygwin) and issue the command '`xtools <target-alias>`', where the target alias is selected from the toolsuite you are using (i.e., arm-linux, mips-linux, mipsel-linux, ppc-linux, etc.), depending on the toolsuite you have installed.

The following simple example shows you quickly how easy it is to get a program compiled, linked and executed in an Instruction Set Simulator (ISS) and debug it using Visual GDB.

```
$ xtools <target-alias> (cr)
In our example, we use the arm-linux.
$ xtools arm-linux
arm-linux$ cd /home/test (cr)
```

In the last step, we assumed that the Microcross test directory either was installed in Cygwin's /home/test or Linux's /home/test.

```
arm-linux$ gcc –g –o pascal.x pascal.c (cr)
```

The program was compiled and linked in one step. You will need to add this file to the Linux filesystem and run it on the target processor using the procedure in Section 3. To debug this simple example, follow the procedure in Section 6.

This concludes the simple example showing how easy it is to compile and link a program with GNU X-Tools. The GNU X-Tools toolsuite is very powerful and can build any conceivable program a developer can code. Now, we need to go into more detail on how the GNU X-Tools work.

9.4.2 GNU X-Tools Toolsuite Description

The `GCC` program is actually a control program that executes the compiler components to produce the desired output, which is usually a compiled and linked executable program image. By manipulating the many `GCC` options and controlling the input file types, the functions of `GCC` are greatly expanded. The `GCC`, however, is but a single component. It is actually a control program that calls other components that perform separate steps to create an executable binary. The components are described as follows and show graphically in Figure 9.1:

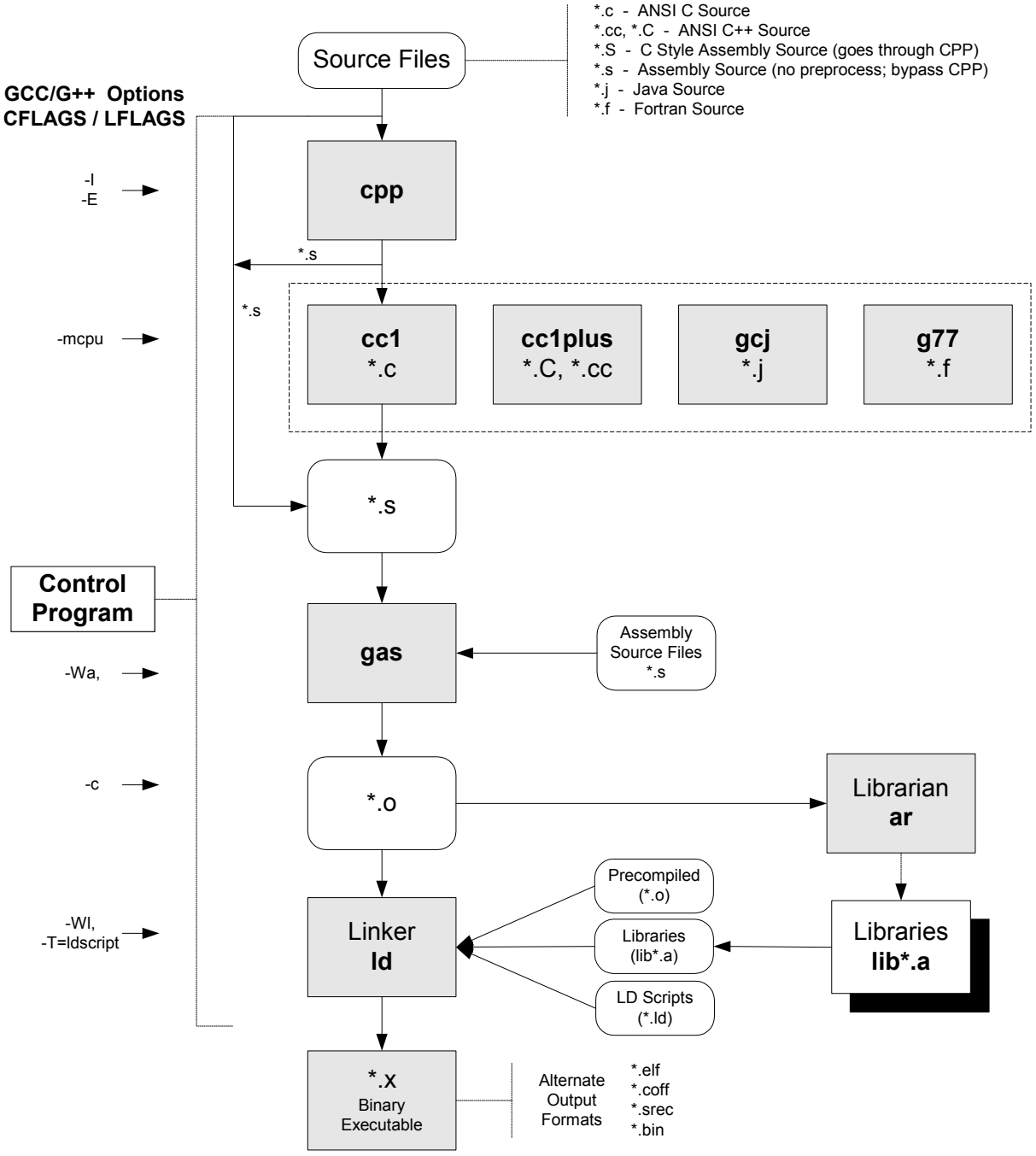
preprocessor
Performed by `cpp`, which is invoked by `GCC`, the preprocessor resolves directives like `'#define'`, `'#include'`, and `'#if'`. Preprocessing establishes the type of source code to process.

compiler
Performed by `GCC`, the compiler pass, which produces assembly language from the input files, and passes the assembly source directly to the assembler phase.

assembler
Performed from `GCC` by `as`, the GNU assembler. It takes the assembly language as input and produces object files with `'.o'` extensions. The assembler output is relocatable object code `('.o')`.

linker
Performed by `ld`, the GNU linker. Linking completes the compilation process, combining all object files (newly compiled, and those specified as input) into an executable file. This step completes the final stage, where the `'.o'` modules are placed in their proper places in the executable file. Library functions that the program refers to are also placed in the file. `GCC` performs this task by internally invoking the linker. `GCC` also cleans up by deleting any object files that it created from source files; however, it does not cleanup any pre-existing object files that you specified on the command line. `GCC` normally invokes all of these compilation steps when converting a C source program into an executable. By using command line options for `GCC`, these steps may be invoked separately or in some combinations. This provides some flexibility when building large programs, or using assembly language sources, or debugging.

Figure 9.1. GNU X-Tools Flow Diagram



9.4.3 Control Program (GCC)

Exactly how the GCC processes any file depends on the file's name. The control program strips the initial part of the name, and then determines how to process the file on the basis of the filename's extension. In each case, the control program passes the file to the appropriate program for preprocessing, compilation, or assembly, and it links all resulting object modules together to produce an executable file. The following table 9.1 shows how the control program recognizes different file types:

Table 9.1. Controlling Compilation / Linking

Input File	Interpretation	Action
`file.c`	C source*	Preprocessed and compiled by `GCC`
`file.C`	C++ source **	Preprocessed by GCC and compiled by `g++`
`file.cpp`	C++ source *	Preprocessed by GCC and compiled by `g++`
`file.CPP`	C++ source **	Preprocessed by GCC and compiled by `g++`
`file.cc`	C++ source *	Preprocessed by GCC and compiled by `g++`
`file.cxx`	C++ source *	Preprocessed by GCC and compiled by `g++`
`file.h`	C or C++ header file	Precompiled header
`file.s`	Assembly language source*	Assembly by `as`
`file.S`	Assembly language source**	Preprocessed and assembled by `as`
`file.o`	Compiled object module*	Passed to `ld`
`file.a`	Object module library*	Passed to `ld`

* Lower case file extension.
** Upper case file extension.

All other files, together with options that `GCC` does not recognize, are passed to `ld`, the linker – either the `'native'` linker supplied by your vendor, or the GNU linker. As a result, almost all linker options are available directly through `GCC`.

In the commands shown earlier in this Section, the `'-l'` and `'-L'` options were actually passed to the linker. In general, `GCC` passes on unrecognized options to the linker, so you can specify linker options without having to invoke the linker separately.

If you're writing C++ code, you can use the `g++` command instead of `GCC`. You're actually getting the same compiler; however, when it is invoked as `g++`, the compiler expects C++ instead of vanilla C source code. In addition, `g++` uses different default libraries.

If you want either to preserve the output of `GCC` at some intermediate stage for debugging purposes or to manipulate the code directly, you can do so. Here is how each stage works:

preprocessing

A `'-E'` option in `GCC` sends the preprocessed code to the standard output, instead of compiling the program.

compilation

To save the assembly language output, run `GCC` with the `'-S'` option. This produces files whose names end with `'.s'` in place of the source file's `'.c'`.

assembly

As we have seen, running with `'-c'` produces object files whose names end with `'.o'`.

We have just finished a long discussion of the many kinds of input that the compiler takes and the different kinds of output that it can provide. Pictures are not always worth a thousand words, particularly when it comes to summarizing a lot of disparate information. But it may help you to remember this information if you view the compiler as a kind of "machine" with different inputs and outputs, as outlined earlier.

The input file's name determines where it goes into the machine. The `'.c'`, `'.S'`, `'.cc'`, `'.cpp'` and `'.C'` files go to the preprocessor, `'.i'` and `'.ii'` files go straight to the compiler, and so on. Compilation options determine which stage of the machine produces output (i.e., how many stages of the machine you run). The `'-S'` means that you stop after the assembler and the output filename ends with `'.s'`. If you keep this picture in mind, the compiler's machinations will not seem so strange; you will stop seeing preprocessing, compilation, assembly, and linking as separate steps and come to see compilation as one big assembly line, for which `GCC` is the production manager.

The following tools listed in Tables 9.2, 9.3, and 9.4 are the main tools for developing projects with a GNU X-Tools Toolsuite.

Table 9.2. GNU X-Tools Compiler and Development Tools

Tool	Description
as	GNU assembler
cpp	C preprocessor
gcc	Optimizing ANSI compliant GNU Compiler Collection (controller of all compilation)
gdb	GNU debugger for source and assembly debugging with command line
gdbtk	Debugger using a graphical user interface called Visual GDB (a.k.a. Insight)
g++	Optimizing ISO tracking GNU C++ compiler
gasp	GNU assembler preprocessor
ld	GNU Linker

Table 9.3. GNU X-Tools Libraries

Tool	Description
libc	Non-restricted ANSI C runtime library *(Newlib for cross-development)*
libgloss	Support library for embedded targets (*board support for cross-development*); (deprecated) will be replaced by a future product
libm	Non-restricted C math subroutine library *(Newlib for cross-development)*
libstdc++	GPL C++ class library, implementing the ISO 14882 Standard C++ library

Table 9.4. GNU X-Tools Binary Utilities

Tool	Utility Description	Target Dependent
addr2line	Converts addresses into file names and line numbers	yes
ar	Creates, modifies and extracts from object code archives	yes
diff diff3 Sdiff	Comparison tools for text files	no
make	Compilation control program	no
nm	Lists symbols from object files	yes
objcopy	Copies and translates object files	yes
objdump	Displays information from object files	yes
patch	Installation tool for source fixes	no
ranlib	Generates index to archive contents	yes
readelf	Displays information about ELF format object files	no
run	Standalone simulator	yes
size	Lists file section sizes and total sizes	yes
strings	Lists printable strings from files	yes
strip	Discards symbols	yes

The Adobe Acrobat (PDF) files listed in Table 9.5 are available on the GNU X-Tools or Cygwin (docs directory) CD-ROM for reference or hard-copy duplication.

Table 9.5. PDF Documentation on CD-ROM

No.	File Name	Description	Pages
1	Bench++.pdf	By Joe Orost, Bench++ is designed to measure the performance of the code generated by C++ compilers, not hardware performance.	15
2	ddd.pdf	Debugging with DDD (Data Display Debugger), v3.3.9, 15 January 2004	234
3	Infoman.pdf	Reading GNU On-Line Documentation	36
4	Make.pdf	How to use GNU Make	166
5	Man-pages.pdf	A Concise Reference Document for all of the Tools in Unix Man Page Format	156
6	MC-Auxiliary-Tools.pdf	Using as Using binutils Using cygwin Using info	340
7	MC-Compiler-Tools.pdf	Using GNU CC Using the C Preprocessor	412
8	MC-Debugging-Tools.pdf	Debugging with GDB Insight, the GNUPro Debugger GUI Interface	260
9	MC-Embedded-Systems.pdf	GNU Tools for Embedded Systems…Everything you want to know about using GNU tools with embedded target processors.	530
10	MC-Libraries.pdf	GNU C Library GNU Math Library GNU C++ iostream Library	294
11	MC-Development-Tools.pdf	Using ld Using make Using diff & patch	350
12	MC-Utilities.pdf	Using as Using ld Using binutils Using make Using diff & patch Using info	682

9.5 Controlling the Tools Using the GCC

The `GCC (GNU Compiler Collection)` control program can be run like the following, producing an executable file from a number of C or C++ source files. Both examples assume that the target toolsuite is installed prior to executing the build commands. The examples shown are with `GCC`, which controls the compilation of C, C++, and assembly files. Also, the user must name the C files with the `'.c'` extension and C++ files with `'.C'` , `'.cpp'` or `'.cc'` extensions.

```
$ target-alias-gcc –options –o <program.out.name> <first.c> <second.c> <third.c>
<fourth.c> …
```

Or

```
$ xtools <target-alias>
<target-alias>$ gcc –options –o <program.out.name> <first.c> <second.c> …
```

Or

```
$ xtools <target-alias>
<target-alias>$ gcc - options one_or_more_source_files.c -o program.out.name
```

Or

```
<target-alias>$ gcc - options -c source_file.c
<target-alias>$ gcc - options file1.o . . . fileN.o -o program
```

All alternatives actually look the same to the compiler parts. The program executed under the name of GCC is just a front that handles options and temporary files and calls the real compiler parts: cpp, the C preprocessor. It takes care of preprocessor directives, such as include files and macro expansions. It also removes comments. The result is a file with the C code, lots of white space and some line-numbering directives that the compiler core can use in warning and error messages.

The '-o' filter argument tells GCC to name the executable file 'program'. If you don't specify an '-o' argument, GCC chooses the default name 'a.out', which is not particularly informative (and would cause multiple executables to overwrite each other). So, most programmers use the '-o' argument to name the program.

Since the inputs to the last example above are all object files, no compilation or assembly is required: GCC always invokes the linker. Using GCC to invoke the linker is preferable to using ld separately, because GCC ensures that the program is linked with the correct libraries and initialization routines.

9.5.1 GCC Options Commonly Used

-c
Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

-o *file*
The '-o' and not '-c' tells GCC to create a linked executable with a user defined 'filename' as its output file. Not specifying '-c' or '-o' will tell GCC to create a default executable named 'a.out' for all cross-compilers and 'a.exe' for the native Cygwin compiler.

-D
One, the '-D' option, acts like '#define' in the source code: it sets the value of a symbol.

```
<target-alias>$  gcc -c -D first=\"info\" -D second example.c
```

The first '-D' option sets 'first' to the string 'info\' (because of the backslashes, the quotation marks actually become part of the symbol's definition). This can be useful for controlling which file a program opens. The second '-D' option defines the 'second' symbol. It happens to set it to the value 1, the default, but you probably don't care; your program just uses an '#ifdef' directive to check whether it's set.

-E
Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output. Input files which don't require preprocessing are ignored.

-S

Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

By default, the assembler file name for a source file is made by replacing the suffix `.c, .i,` etc., with `.s.`

Input files that don't require compilation are ignored.

-v

Print (on standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

-###

Like -v except the commands are not executed and all command arguments are quoted. This is useful for shell scripts to capture the driver-generated command lines.

-pipe

Use pipes rather than temporary files for communication between the various stages of compilation. This fails to work on some systems where the assembler is unable to read from a pipe; but the GNU assembler has no trouble.

--help

Print (on the standard output) a description of the command line options understood by gcc. If the `'-v'` option is also specified then `'–help'` will also be passed on to the various processes invoked by gcc, so that they can display the command line options they accept. If the `'-Wextra'` option is also specified then command line options which have no documentation associated with them will also be displayed.

--target-help

Print (on the standard output) a description of target specific command line options for each tool.

--version

Display the version number and copyrights of the invoked GCC.

-Uname

Some symbols – usually those identifying the type of computer system you're compiling on – are automatically defined by the compiler. If you want to suppress one of these symbols, use the `'-Uname'` option. This is equivalent to putting `'#undef name'` at the beginning of each source file.

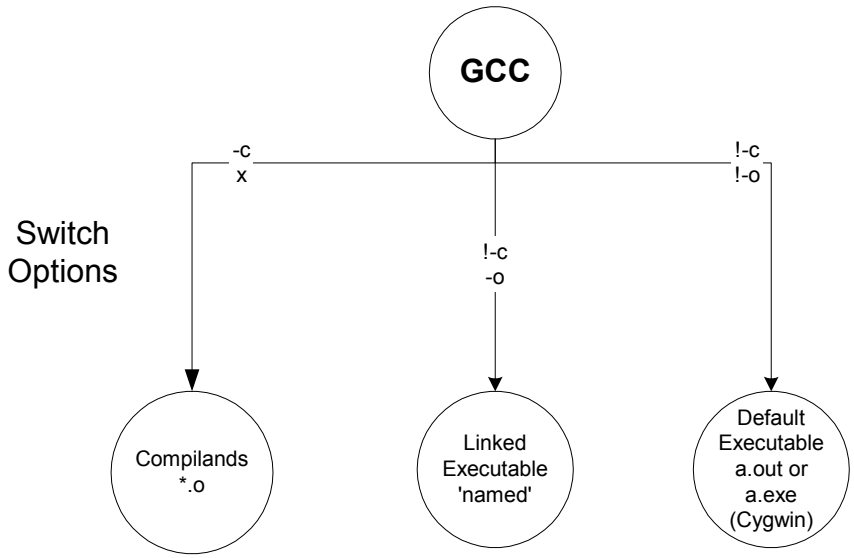
-I

If you have an include file in a non-standard directory, specify this directory in a `'-I'` option. For example, suppose you have two directories for code, `'/usr/src'` for source files and `'/usr/headers'` for header files. While you're compiling in the `'/usr/src'` directory, you can tell `GCC` where to find the header files through the command:

```
<target-alias>$ gcc –I../headers example.c  (cr)
```

Figure 9.2 below summarizes the three primary modes of `GCC` with switch actions.

Figure 9.2. Three Primary Modes of GCC



9.6 Controlling Linker from GCC

Another common compilation option is `'-l'`, which specifies a library and `'L'`, which specifies the path to a non-standard library. Important, `'-l'` must be specified at the end of the command after the source/object filenames, while all other options are specified before the source/object filenames. The examples shown are with `GCC`, but the user can substitute `g++` in place of `gcc` along with the appropriate C++ files and libraries needing linked. Here's a typical command; the inputs are object files, so the command just runs the linker.

```
<target-alias>$ gcc –o example –L/src/local/lib main.o first.o –lm (cr)
```

This command tells `GCC` to look for libraries first in `'/src/local/lib' path`, then in the standard location. If someone has put a local version of a library in `'/src/local/lib'`, that version is used in preference to the standard version in `'/usr/lib'`. The `'-lm'` specifies the library name -- actual library name is `'libm.a'`. The prefix and suffix to the library name can be dropped on all library names. The naming convention for GNU libraries is lib<name-of-lib>.a (static libs) and lib-<name-of-lib>.so (shared libs).

Unix / Linux linkers search libraries in the order in which they occur on the command line and only resolve the references that are outstanding at the time the library is searched. Therefore, the order of libraries and object modules on the command line can be critical (this is why `'-l'` options come after the filenames). Consider the command:

```
<target-alias>$  gcc –l mine first.c (cr)
```

This command searches for the library file `'lib`**`mine`**`.a'` to resolve any function references needed for linking; however, the linker has not yet processed the object module for `'file4.o'` (created by the `GCC` command and normally deleted if compilation and linking are successful). Therefore, there are no outstanding function references, and the library search has no effect. If the program needs this library, the `GCC` command produces `'Undefined symbol'` messages during the loading phase, and the linker does not produce an executable file. To perform this compilation correctly, enter the command:

```
<target-alias>$  gcc first.c –lmine (cr)
```

Now the loader searches the library after processing `'first.c'` and is able to resolve any references requiring this library.

When compiling a C program, you often don't need to list any libraries explicitly on the command line. The `GCC` automatically searches the system call library, I/O library, and run-time initialization routines. If you use any math routines, you'll need to search for the math library (`'-lm'`); if you're compiling C++ code, you need to include the C++ libraries (`'-lstdc++'`).

9.7 Compilation Options

The following sections discuss other important options that are available with the `GCC`. There are many, many other options – perhaps several hundred all-together – that control various details of compilation and optimization. The chances are that you will never need these, but it won't hurt to familiarize yourself with the complete documentation for `GCC`. Complete coverage of `GCC` options is provided in *Using and Porting GNU CC* [reference 20].

9.7.1 Displaying compiler behavior

The `'-v'` (verbose) option prints the compiler's version number and complete details about how each pass is executed. This option is particularly useful for finding out exactly which options your program is being linked with.

9.7.2 C Language Options

We'll start by listing a few options for controlling the warning messages that `GCC` produces. There are many options for controlling warnings; it's possible to request (or inhibit) many warning messages on a per-message basis. We're not sure that's really useful; we'll limit ourselves to a few options that control large group of messages:

`-w`
Suppress all warning messages.

`-W`
Produce some additional warning messages about legal (but questionable) coding practices.

`-Wall`
Produce even more warning messages about questionable coding practices.

`-Wtraditional`
Produce warning messages about code that is legal in both the 'Kernighan and Ritchie' and ANSI definitions of the C language, but behaves differently in each case.

`-Werror`
Make all warnings into errors; that is, don't attempt to produce an object file if a warning has occurred.

Now, we'll discuss how to control various features of the C and C++ languages. There are basically three options to worry about: `'-traditional'`, `'-ansi'`, and `'-pedantic'`. In most cases, it's fairly easy to tell which you want. Older C code – code that pedants the standard – should be compiled with `'-traditional'`. Newer code that has been written to conform to the ANSI standard should be compiled with `'-ansi'`. Either option can accept prototypes, where you specify the arguments on the same line as the function name, as in `'func (char *arg)'`.

Note that the ANSI C standard and "traditional" (Kernighan and Ritchie) C both define the behavior of the preprocessor – either explicitly or implicitly. Therefore, the options listed below affect both `cpp` (C preprocessor) and `GCC`:

`-traditional`
Supports the traditional C language, including lots of questionable, but common, practices. The traditional option also supports all of the FSF's extensions to the C language.

`-ansi`
Supports the ANSI C standard, though somewhat loosely. The FSF's extensions are recognized, except for a few that are incompatible with the ANSI standard. Thus, ANSI programs compile correctly, but the compiler doesn't try too hard to reject non-conformant programs, or programs using non-ANSI features.

`-pedantic`
Issues all the warning messages that are required by the ANSI C standard. Forbids the use of all the FSF extensions to the C language and considers the use of such extensions errors. As the `GCC` manual points out, `'-pedantic'` is not a complete check for ANSI conformance, it only issues errors that are required by the ANSI standard.

9.7.3 Preprocessor Options

The following set of options control the `cpp` preprocessor from the command line:

`-M`
Read the source files; figure out which files they include, and output lists of dependences for make. There is one dependency list for each source file. The dependency lists are sent to standard output, and compilation doesn't proceed past preprocessing (i.e., `'-M'` implies `'-E'`). This option can make it much easier to generate correct makefiles.

`-C`
The preprocessor normally deletes all comments from the program. With `'-C'`, it doesn't. This flag may be useful in conjunction with `'-E'` when you are trying to make sure that the preprocessor is doing what you intended. In such cases, leaving you comments in may be handy. The `'-C'` option doesn't automatically imply `'-E'`, but `GCC` won't let you use `'-C'` on the command line unless `'-E'` is also present.

9.7.4 Options to Specify Libraries, Paths and Startup Files

The following options are common for embedded developers, but not in native environments:

`-nostartfiles`
Don't use the standard system startup files when linking. Normally the `'crt0.o'` file gets linked in as the standard start file; however, most embedded developers will need to replace the standard start file with a custom start file. When used with GCC, add the correct option `'-Wl,-nostartfiles'` and add the custom startfile with appropriate entry symbol to the linker script (e.g., `ENTRY(_start)`). To learn more about how to create linker scripts and startfiles, read the GNU X-Tools Training Guide or the Microcross Visual X-Tools User Guide.

`-nodefaultlibs`
Do not use the standard system libraries when linking. Only the libraries you specify will be passed to the linker. The standard startup files and used normally, unless `'-nostartfiles'` is used. The compiler may generate calls to memcmp, memset, and memcpy for System V (and ANSI C) environments or to bcopy and bzero for BSD

environments. These entries are usually resolved by entries in libc. These entry points should be supplied through some other mechanism when this option is specified.

-nostdlib

When linking, this option tells the linker do not use the standard libraries or startup files. This option is useful when you want to provide your own libraries, overriding the default libraries and use your own custom startup files. When used with GCC, add the correct option '-Wl,-nostdlib'. If you plan on augmenting the standard libraries with your own libraries, then do not use this option. Use a 'GROUP' and 'SEARCH_DIR' statement with appropriate arguments in the linker script – see the Visual X-Tools IDE User Guide or the GNU X-Tools Training Guide for examples on how to create linker scripts and startfiles.

-nostdinc

Do not search the standard system directories for header files. Only the directories you have specified with '-I' options and the current project directory are searched. By using both '-nostdinc' and '-I-' you can limit the include file search path to only those directories you specify explicitly.

-static

Link only to static libraries, not shared libraries. When used with GCC, add the correct option '-Wl,-static'. In GNU X-Tools, static link is the default setting.

-shared

If shared libraries are available, use them wherever possible, rather than static libraries. When used with GCC, add the correct option '-Wl,-shared'.

-I*dir*

Add the directory *dir* to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one '-I' option, the directories are scanned in left-to-right order; the standard system directories come after.

-I-

Any directories you specify with '-I' options before the '-I-' option are searched only for the case of '#include "file"'; they are not searched for '#include <file>'. If additional directories are specified with '-I' options after '-I-', these directories are searched for all '#include' directives. Ordinarily all '-I' directories are used this way. In addition, the '-I-' option inhibits the use of the current directory (where the current input file came from) as the first search directory for '#include "file"'. There is no way to override this effect of '-I-'. With '-I.' you can specify searching the directory that was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory. '-I-' does not inhibit the use of the standard system directories for header files. Thus, '-I-' and '-nostdinc' are independent.

-L*dir*

Add directory *dir* to the list of directories to be searched for '-l'.

`-B`*prefix*

This option specifies where to find the executables, libraries, include files, and data files of the compiler itself. The compiler driver program runs one or more of the subprograms `'cpp'`, `'cc1'`, `'as'`, and `'ld'`. It tries prefix as a prefix for each program it tries to run, both with and without `'machine/version/'`. For each subprogram to be run, the compiler driver first tries the `'-B'` prefix, if any. If the name is not found, or if `'-B'` was not specified, the driver tries two standard prefixes, which are `'/usr/lib/gcc-lib/'` and `'/usr/local/lib/gcc-lib'`. If neither of those results in a file name that is found, the unmodified program name is searched for using the directories specified in your `'PATH'` environment variable. `'-B'` prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into `'-L'` options for the linker. They also apply to includes files in the preprocessor, because the compiler translates these options into `'-isystem'` options for the preprocessor. In this case, the compiler appends 'include' to the prefix. The run-time support library file `'libgcc.a'` can also be searched for using the `'-B'` prefix, if needed.

`-specs=`*file*

Process *file* after the compiler reads in the standard `'specs'` file, in order to override the defaults that the `'gcc'` driver program uses when determining what switches to pass to `'cc1'`, `'cc1plus'`, `'as'`, `'ld'`, etc. More than one `'-specs='` file can be specified on the command line, and they are processed in order, from left to right.

9.7.5 Debugging and Profiling Options

These options request the compiler to create additional code and an expanded symbol table for the various profilers and debuggers (`dbx`, `prof`, `gprof`, and the branch count profiler). They are extremely helpful for debugging and tuning code under development, but should not be used for production release versions of your program.

`-p`

Link the program for profiling with `prof`. When you execute a program compiled with this option, it produces a file named `'mon.out'` that contains program execution statistics. The profiler `prof` reads this file and produces a table describing your program's execution.

`-pg`

Link the program for profiling with `gprof`. Executing a program compiled with this option produces a file named `gmon.out` that includes execution statistics. The profiler `gprof` reads this file and produces detailed information about your program's execution. For example, the following command compiles the file `'program.f'`, generating code for profiling with `gprof`:

```
<target-alias>$ gcc -pg program.f
```

`-g`

Generate a symbol table for debugging. The `'-g'` option enables debugging with the GNU debugger, `GDB` or `Visual GDB`. The symbols used with standard GDB are called stabs. When compiling with Microcross' cross-compilers, set the `'-g'` option to produce the stabs symbols; you may use `'-gstabs'` as a verbose switch if you like. Other symbol options include `'-gcoff'`, `'-gdwarf'` and `'-gdwarf2'`; the `'-gcoff'` symbols are not supported in Microcross GNU cross-compilers; however, `'-gdwarf'` and `'-gdwarf2'` are supported in most target CPU toolsuites.

IMPORTANT NOTES: To perform assembly debugging of straight assembly language files (`.s` or `.S` files), you must use `'-g'` or `'-gstabs'` because the assembler will not generate any other symbol formats. The C/C++ compiler, however, will generate appropriate symbols for either `'-gdwarf'` or `'-gdwarf2'` in addition to stabs. When compiling with any debug option, do not use optimization options. You cannot optimize code and debug it correctly using the GNU tools.

9.7.6 Optimization

GNU X-Tools GCC incorporates a sophisticated optimizing compiler; and for most target systems, it usually generates faster code than with the native compiler. Any '-Ox' optimization can be used with any '-fx' option (e.g., gcc -Os –funroll-loops –ffast-math –o outfile.x infile.c). Here are the most commonly used compilation options:

-O0
No optimization. This is the default. With optimization turned off, GCC tries to generate code that is easy to debug; you can set a breakpoint between any two statements and modify variables, and the program will behave exactly as it should. The GCC also tries to generate code quickly.

-O1
The compiler tries to moderately to reduce both the size of the compiled code and the execution time. Compilation is slower than with '-O0' and requires more memory.

-O2
Enables more optimizations than '-O1'. Compilation time is even slower; the resulting code should be even smaller and faster.

-O3
Enables more optimizations than '-O2'. Try hardest of all to produce fast assembly code. Note that the emphasis is on fast: the resulting code may take much more room in memory because certain functions may be placed in-line and loops may be unrolled (as if each iteration in the for loop were written out independently).

-Os
Try to produce code that is small. The emphasis is on size, not speed. This option uses many of the same optimization algorithms as '-O2', but with a different emphasis.

-ffast-math
Make floating-point arithmetic optimizations that violate the ANSI or IEEE standards. Code compiled with this option may yield incorrect results, but it will be slightly faster. Be sure to test your program thoroughly.

-finline-functions
Expand all “simple functions” into their callers. The compiler gets to decide whether any function is “simple” or not. Inline expansion is a two-edged sword; it can make a program faster (by eliminating calling overhead) or slower (by making instruction cache performance worse).

-fno-inline
Inhibit all inlining, even inlining that is requested by the inline keyword in the source code. The GCC performs inlining according to statements in the source code with both '-O1' and '-O2'; the keyword is ignored if optimization is not in effect.

-funroll-loops
On some CPUs, loop unrolling can be a very important optimization. It minimizes loop overhead and creates many opportunities for further optimizations. With the '-funroll-loops' option, GCC unrolls all loops that have a fixed iteration count known at the time of compilation. Loop unrolling is another risky optimization; it can improve

performance, but it can also penalize performance significantly: the object file becomes larger, which can significantly hurt cache performance, and compile time also increases.

9.7.7 Passing Options to the Assembler or Linker

The GCC allows you to pass options directly to the assembler or linker when they are involved:

`-Wa,options`
Pass the 'options' to the assembler

`-Wl,options`
Pass the 'options' to the linker

In both cases, the 'option-list' is just a list of options recognized by the assembler or the linker. There must not be any spaces in the list; options in the list are separated by commas.

Here is an example that is both instructive and useful: producing a listing of the assembly language generated, together with C source listings. To do this, we need to pass the '-alh' options to the assembler (generate listings of assembly code and high-level source); we also need to pass the '-L' option to the assembler (retain local labels). And we need GCC's '-g' option (generate additional symbols for debugging; the additional symbols tell the assembler where to find the source code). The resulting command looks like this:

```
<target-alias>$  gcc -c -g -Wa,-alh,-L source.c (cr)
```

Listings that include both assembly and source code are interesting from two standpoints. You may want to see how your code has been compiled; this is instructive, whether or not you are optimizing and even if you are not interested in assembly level debugging. What is more important, though, is that you can generate a C / assembly listing for optimized code. This can be very helpful for debugging under optimization. The big problem with debugging optimized code is that there is no longer a simple mapping from your source code into assembly language; therefore, the debugger cannot single step and perform symbolic debugging. With a listing, you can find out exactly what the compiler did to your code and get a much better idea of what the code is doing.

9.8 Using the GNU Assembler

The GNU assembler is really many assemblers folded into one (or many different programs with the same name, depending on how you look at it). You can usually ignore the assembler; the compiler invokes it automatically and is usually able to specify everything the assembler needs to know about your environment. In rare cases, you may need to ask for an assembly option explicitly; in these cases, you will need to run the assembler, as, a separate program or use GCC's '-Wa' option to pass additional options to the assembler. The assembler arguments must be separated from each other (and the `-Wa') by commas. For example:

```
<target-alias>$ gcc -c -g -O -Wa,-alh,-L file.c (cr)
```

This above example emits a listing to standard output with high-level and assembly source.

Usually you do not need to use this `-Wa' mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler. You can call the GNU compiler driver with the `-v' option to see precisely what options it passes to each compilation pass, including the assembler.

Next, we discuss what the assembler does. We will not discuss the assembly language itself. The FSF's documentation explains the general syntax of assembly language, but refers you to the vendor's architecture manual for CPU dependent details: overall architecture, instruction set, etc..

The assembler takes a program written in an assembly language and produces an object module. By convention, assembly language programs have the extension `'.s'`. If no errors occur during assembly and if the object module contains no references to external (imported) symbols, the assembler makes the file executable and names it `'a.out'`. If the object module includes references to external symbols, `'a.out'` is not an executable. The linker is able to link this object module with other modules to produce an executable program.

To invoke the assembler, enter the command:

```
<target-alias>$  as -options <list-of-source-files> (cr)
```

Where `'list-of-options'` is a series of assembly options and `'list-of-source-files'` is one or more assembly language files (`.s`) -- `.S` (capital S) requires `GCC` to direct the source file to the preprocessor and then on to the assembler . Unlike most Unix assemblers, the GNU assembler can work on several files at a time.

The assembler has many options; most of them are architecture-specific and are used to describe the target processor more precisely. These options will be important to you if you are cross-compiling; check the FSF's manual for more details. Each appendix in this user guide briefly discusses the target dependent compiler and assembler options.

The following are a set of options and controls that are generally useful for invoking all GNU assemblers:

Common assembler switch options to all Targets

as	[-a[cdhlns][=file]] [-D] [--defsym sym=value]
	[-f] [--gstabs] [--help] [-I dir][-J][-K][-L]
	[--keep-locals] [-o objfile] [-R] [--statistics]
	[-v] [-version] [--version] [-W] [-w] [-x] [-Z]
	[-mbig-endian | -mlittle-endian] (if implemented)
	[-mfpa10 | -mfpa11 | -mfpe-old | -mno-fpu] (if implemented)
	[-EB | -EL] (if implemented)
	[-O]
	[-O | -n | -N]
	[-b] [-no-relax]
	[-nocpp] [-G num] [-mcpu=CPU] (if implemented)
	[--trap] [--break]
	[--emulation=name]
	[-- | files...]

These assembler switch options are discussed in depth within the manual, *MC-Utilities.pdf* (docs directory on GNU X-Tools CD-ROM and in Cygwin/docs). In addition, each assembler has target specific switch options. The target-specific switch options are located in the Appendices of this User Guide.

Unlike older assemblers, `as` is designed to assemble a source program in one pass of the source file. This has a subtle impact on the `'.org'` directive. These options enable listing output from the assembler. By itself, `` `-a' `` requests high-level, assembly, and symbols listing. You can use other letters to select specific options for the list: `` `-ah' `` requests a high-level language listing, `` `-al' `` requests an output-program assembly listing, and `` `-as' `` requests a symbol table listing. High-level listings require that a compiler debugging option like `` `-g' `` be used, and that assembly listings (`` `-al' ``) be requested also.

Use the `-ac' option to omit false conditionals from a listing. Any lines which are not assembled because of a false '.if' (or '.ifdef', or any other conditional), or a true '.if' followed by an '.else', will be omitted from the listing. Use the `-ad' option to omit debugging directives from the listing.

Once you have specified one of these options, you can further control listing output and its appearance using the directives '.list', '.nolist', '.psize', '.eject', '.title', and '.sbttl'. The `-an' option turns off all forms processing. If you do not request listing output with one of the `-a' options, the listing-control directives have no effect. The letters after `-a' may be combined into one option, e.g., `-aln'.

-a[cdhlmns]

Turn on listings, in any of a variety of ways:

-ac	Omit false conditionals
-ad	Omit debugging directives
-ah	Include high-level source
-al	Include assembly
-am	Include macro expansions
-an	Omit forms processing
-as	Include symbols
=file	Set the name of the listing file

You may combine these options; for example, use '-aln' for assembly listing without forms processing. The '=file' option, if used, must be the last one. By itself, '-a' defaults to '-ahls'.

-D	**Ignored***
--defsym sym= value	Define the symbol sym to be value before assembling the input file. value must be an integer constant. As in C, a leading 0x indicates a hexadecimal value, and a leading 0 indicates an octal value.
-f	"Fast"—skip white space and comment preprocessing (assume source is compiler output). Warning: if you use `-f' when the files actually need to be preprocessed (if they contain comments, for example), as does not work correctly.
--gstabs	Generate stabs debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it.
--help	Print a summary of the command line options and exit.
-I *dir*	Add directory, 'dir', to the search list for '.include' directives. You may use -I as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, as searches any `-I' directories in the same order as they were specified (left to right) on the command line.
-J	Don't warn about signed overflow
-K	Issue warnings when difference tables altered for long displacements.
-L	Keep (in the symbol table) local symbols, starting with L.
--keep-locals	Keep (in the symbol table) local symbols. On traditional 'a.out' systems these start with L, but different systems have different local label prefixes.
-M or --mri	This changes the syntax and pseudo-op handling of as to make it compatible with the ASM68K or the ASM960 (depending upon the configured target) assembler from Microtec Research.
--MD	as can generate a dependency file for the file it creates. This file

	consists of a single rule suitable for make describing the dependencies of the main source file.
-o	objfile Name the object-file output from as objfile. By default it has the name `a.out' (or `b.out', for Intel 960 targets only). You use this option (which takes exactly one filename) to give the object file a different name.
-R	Fold the data section into the text section.
--statistics	Print the maximum space (in bytes) and total time (in seconds) used by assembly.
--strip-local-absolute	Remove local absolute symbols from the outgoing symbol table.
-v -version	Print the as version.
--version	Print the as version and exit.
-W	Suppress warning messages.
-w	Ignored*
-x	Ignored*
-z	Generate an object file even after errors.
-- | files...	Standard input, or source files (files ...) to assemble.

* This option is accepted for script compatibility with calls to other assemblers.

9.9 Using the Linker

The ld linker combines several object modules and libraries into a single executable file. It resolves references to external variables, external procedures, and libraries, creating a complete, self-sufficient program. You never need to invoke ld explicitly. In most cases, it is simpler to use the GCC command to link files, even if you do not have any source files to compile. The GCC guarantees that certain libraries will be present in the proper order even if they are not listed on the command line. If you use ld as a linker, you need to mention these libraries explicitly.

9.9.1 Invoking ld

The rules for invoking ld, if you must do so, are the same as for GCC or as. The basic ld command is as follows:

```
<target-alias>$ ld <list-of-options> <list-of-files-and-libraries> (cr)
```

Where 'list-of-files-and-libraries' is a series of filenames and library specifications. To include a library in this list, use the notation '-lname', where the name of the library file is either '/lib/libname.a' or '/usr/lib/libname.a'. The linker processes the 'list-of-files-and-libraries' in order. When it reaches a library, it extracts only those modules that it currently needs to resolve external references. Consequently, the position in which libraries appear in this list is important. For example, the command:

```
<target-alias>$ ld prog1.o -lm prog2.o (cr)
```

results in an 'Undefined symbol' message if 'prog2.o' refers to any programs in the library '/usr/lib/libm.a' – unless you happen to be lucky and 'prog2.o' only uses routines that the linker extracted for the sake of 'prog1.o'. Note that libraries may refer to other libraries; thus, the command:

```
<target-alias>$ ld prog1.o -lat -lfo (cr)
```

leads to 'Undefined symbol' messages if the 'fo' library requires any routines from 'at'.

This situation is more complex for a user-generated library. Such a library should contain an index, so that the linker can find each module regardless of its order within the library. Some systems always generate an index

when you create or modify the library with the `ar` command (the GNU `ar` does this). On other systems you have to put in the index yourself by using the `ranlib` command.

If you want to create an executable file, the beginning of the first file in the list-of-files must be the program's entry point. This is not the same as the apparent entry point to your C source program. Before your program begins executing, the computer must execute a standard run-time initialization routine. To ensure that this is in place, `'/lib/crt0.o'` must be the first file in the `'list-of-files-and-libraries'`. This ensures that this initialization routine is linked to your program.

Alternatively, you can link by using the `GCC` command without any C source files. When `GCC` invokes the linker, it automatically adds `'crt0.o'` and many other libraries in the proper place. For example, the command `'gcc exp.o'` generates the following `ld` command:

```
<target-alias>$ ld -dc -dp -e start -X o -o a.out /usr/<target-alias>/lib/crt0.o -lc
```

In this command, the run-time initialization module `'/usr/<target-alias>/lib/crt0.o'` appears explicitly, in addition to requests to resolve references to the C library (the general runtime library). You can see what `ld` command is generating when you compile a program on your system, by invoking `GCC` with the `'-v'` (verbose) option.

9.9.2 Linker Options

The `GCC` passes any options it does not recognize to the linker. The most important options can therefore be placed directly on the `GCC` command line. These options are:

`-o name.x`
Instead of naming the executable output file `'a.out'` or `'a.exe'` (native builds under Cygwin), it names it `'name.x'`. Other popular naming extensions for embedded applications include: `'name.elf'`, `'name.coff'`, `'name.srec'`, and `'name.bin'`. Microcross has standardized all of its cross-executable names to the `'.x'` extension.

`-lname`
Link the program to the library named `libname.a`. The linker looks in the directories `'/lib'` and `'/usr/lib'` to find this library. Note that the GNU linker truncates the name's prefix and extension (i.e., `'lib'` and `'.a'` are not necessary for `'libname.a'` as a linked library file. If you create your own libraries to link into your program, you will need to name them `'libname.a'` and link in as `'-lname'`.

`-Ldir`
To find any libraries, look in the directory `dir` before looking in the standard library directories `'/lib'` and `'/usr/lib'`.

`-s`
Remove the symbol table from the executable output file. This makes the output file significantly smaller, but makes debugging nearly impossible. Therefore, this option should not be used until the program works successfully. Note that using the program `strip` has the same effect.

`-x`
Remove all local symbols from the output file. Global symbols (subprograms and common block names) remain in the output file. This reduces the object file's size. Ignored unless `'-s'` is specified.

`-n`

Make the text segment read-only.

-r

Create an object file that can be included in further linking runs (i.e., further passes through ld). Among other things, this inhibits 'Undefined symbol' messages (because the symbols may be defined in a later ld pass) and prevents ld from creating common storage immediately. If you wish to create common storage at this time, use the '-d' option also.

-e name

Use the symbol name as the entry point to the executable program. By default, the entry point is the beginning of the first object module. The GCC automatically links your object files with a run-time initialization module ('/usr/lib/crt0.o') that starts your program and provides its initial entry point. If you run the linker separately, you must either put '/usr/lib/crt0.o' at the start of your object files, or provide your own entry point.

-M

Produce a load map that shows where each function is located in the resulting object file, where each section begins and ends, and what the value of each global symbol is. This option is usually used with GCC as follows: '-Wl,-M=<mapfile-name>', where '<mapfile-name>' can be any name you specify.

-b format

Read object modules in the given format. To get a list of formats that ld understands, give the command 'objdump -I' and near the bottom of the options list are the supported targets and architectures. This can be helpful in some cross-development situations. The '-b' option applies to all object files and libraries following it on the command line, until another '-b' option appears. In theory, you can use this feature to link objects from several different formats into a single executable.

-oformat format

Create object modules in the given format. Again, 'objdump -I' gives you a list of formats that ld understands. The ld is configured to produce the most reasonable output format for its target machine. Its assumptions about what is "reasonable" are probably true about 99.99 percent of the time. But there may be special-purpose situations in which you would want another output format.

Here is an example of a customized ld command:

```
<target-alias>$ ld -r -o bigofile.o prog1.o prog2.o -lmylib (cr)
```

This command links the files 'prog1.o' and 'prog2.o' and the library file '/usr/lib/libmylib.a'. The resulting file is named 'bigofile.o'; it can be linked further and may still contain unresolved references.

9.9.3 Linker Scripts

One advanced feature of the GNU linker is its ability to work from scripts written in its own command language. If you are a true masochist, you might be able to avoid running GCC altogether; you might be able to implement your own compiler as a linker script. There are a few situations in which you would actually need a linker script, but you should be aware that they exist for purposes like:

- Gaining tight control over the format of the output file – perhaps so an embedded application will fit into the smallest possible ROM, perhaps to optimize link order.

- Supporting an object format that `ld` does not provide – perhaps an object format of your own design, or an object format for some special-purpose operating system.

Microcross recommends starting with the default linker script and modifying it to meet your needs. To extract the standard linker script into a file, type the command:

```
<target-alias>$ ld –verbose >linker.ld (cr)
```

This command will create a file named `'linker.ld'`, and you will need to open this file in an editor to delete the header lines before and including`'======'` and the footer single line `'======'`. Then the script file can be saved or copied into a source directory and be used with the `GCC` option `'-Wl,-T,linker.ld'` to replace the standard linker script. Now, you can modify the linker script with confidence that you have a working startup script.

9.9.4 Link-Order Optimization

If you have done a lot of development work, you have probably noticed that the order in which you link your files can have a significant effect on performance. By changing the link order, you are changing the way the executable file lies in the instruction cache. The cache is a fast area of memory that stores pages of instructions so that the processor does not have to go back to slower parts of memory (or even worse, the disk) for every new instruction. Certain link orders minimize instruction-cache miss. The effect usually is not large, but in pathological cases (a really bad link order on a machine that is very sensitive to cache miss), link optimization can speed up runtime by 50 percent.

Unfortunately, not much can be said about link-order optimization. There are few rules, if any; and all the rules have many exceptions. In general, it is a good idea to place modules that call each other near each other on the command line. The reasoning behind the heuristic is simple: if function A makes many calls to function B, and both A and B can fit into the cache simultaneously, you will not pay a penalty for cache miss. Your best chance of fitting both functions into the cache simultaneously occurs when they are located next to each other in the object file. The `'-M'` option, which produces a load map, shows you how the object file is arranged; it will help you investigate cache performance.

Normally, rearranging the order of the object modules on the command line is sufficient for experimenting with link-order optimization. However, you can get very fine control over your executable file by writing a linker script. If you have a thorough knowledge of your target machine's architecture, you may be able to use this to your advantage – though you will probably reach the point of diminishing returns fairly quickly.

9.9.5 The C Runtime (crt0)

To link and run C or C++ programs, you need to define a small module, usually written in assembly as `'crt0.s'`, but sometimes written as a C file as `'crt0.c'`, to initialize hardware using C conventions before calling main. There are some examples available in the sources of Newlib C library; perform a simple search through the source tree for `'crt0.s'` code, as well as examples of system calls with sub-routines.

To write your own `'crt0.s'` or `'crt0.c'` module, you need specific information about your target – see `'MC-Embedded-Systems.pdf'` in the GNU X-Tools CD docs directory (also in Cygwin/docs) for details on how to create the C runtime environment.

9.10 Object Translation (ELF to Binary, SREC, etc.)

Most of the GNU X-Tools toolsuites produce an Extended Linker Format (ELF) object file as the default format, and it can be used during debug and testing; however, the final code needs to be stripped and translated to another

format. It is beyond the scope of the user guide to explain all of the various formats, which include COFF, ECOFF, binary, srec, tekhex, ihex, symbolsrec, etc..

Here is a quick example of how to use objcopy; refer to the binutils documentation in the docs directory on the distribution CD or in the Cygwin/docs directory for details on how to use. From the Shell, issue the following commands (substitute your target alias and bfdname as necessary).

e.g. (notional only):

```
$ xtools arm-linux (cr)
arm-linux$ cd /home/test (cr)
arm-linux$ gcc –g –o div.elf div.c (cr)
arm-linux$ objcopy –S –O binary div.elf div.bin (cr)
```

Note: The '-S' strips all symbols (including the debugging symbols put in the file by using the '-g' option) out to make the object as small as possible. The '-O binary' specifies the output target format. The input file is 'div.elf' and the output file is 'div.bin'.

9.11 Creating/Updating Libraries

The command ar creates libraries (or archives) of object modules. They are similar to the Unix / Linux utilities with the same names, except that you do not need a separate ranlib. This section gives a brief description of how to use these utilities in the GNU X-Tools command shell.

In naming a library for use with GNU X-Tools, always use the prefix 'lib' and suffix extension 'a' with your static library name; this is the GNU convention and is necessary for GCC to automatically scan the library at link time, so the syntax for linking a particular library, say libm.a the math library, is -lm.

To create a new library, use the ar command, as follows:

```
<target-alias>$ ar -rs lib<name>.a list-of-files (cr)
```

The option 'r' indicates that the command ar should add the files named in the 'list-of-files' to the library named 'name', creating a new library if necessary. If a file is mentioned twice in the 'list-of-files', ar includes it in the archive twice. The s' option tells ar to produce an index for the archive; this is the function that ranlib would perform. If you include the 's' option whenever you create or modify a library, you will not need to use ranlib.

To update a library, use the command:

```
<target-alias>$ ar -rus lib<name>.a list-of-files (cr)
```

This compares the dates of any listed files with the version of the file in the library. If the file in 'list-of-files' is more recent than the version contained in the library, ar substitutes the newer version for the older version. The's' option updates the library's index.

To delete one or more files from a library, use the command:

```
<target-alias>$ ar -ds lib<name>.a list-of-files (cr)
```

This option deletes all the files found in 'list-of-files'.

To extract one or more files from a library, use the command:

```
<target-alias>$ ar -x lib<name>.a list-of-files (cr)
```

This does not modify the library file itself. It extracts the files named in the 'list-of-files' from the library, regenerating them in the current directory with their original names. Normally, the timestamp of the extracted files is the time at which ar recreated them. If you use the option 'xo' instead of 'x', ar sets the timestamp of the extracted files to the time recorded in the archive.

You can still create an ordered (index-less) library with ar and invoke ranlib as a separate step if you want, which is the convention of older Unix Systems. However, there is no longer any good reason for doing that when using GNU tools.

9.12 GNU Libraries

If you are familiar with Unix / Linux and C programming, the libraries you will find in the GNU X-Tools development environment should not confuse you. The libraries you expect will all be there: standard I/O, the math library, the strings library, etc.. The libraries are ANSI C and POSIX compliant. Moreover, there are many functions that Unix / Linux programmers expect, but are not specified by either of these standards.

Some C functions have been standardized and found in the GNU environment, but system calls and math libraries are not the same as those found in a typical DOS / Windows environment. You will have to re-learn these functions. A good resource for learning the GNU Libraries is on the GNU X-Tools CD 'Docs' directory: 'MC-Libraries.pdf' and 'Math_lib.pdf'.

Appendix 1. GNU X-Tools ARM Linux Toolsuite

MODEL (target alias):	ARM -- Advanced RISC Machines (alias: arm-linux)
MFR:	Multiple Manufacturers: ARM, Atmel, Freescale, Intel, NetSilicon, Philips, Samsung, Sharp, and many others
CPU TYPE:	RISC Von Neumann, 32-Bit Fixed Instructions, 16-Bit Thumb mode
GP REGS/SIZE:	16 / 32-Bit
ADDR SPACE:	64MB Linear -- Endian: Selectable
CYCLE RATE:	Core Dependent
APPROX INSTRUCTION THROUGHPUT (PIPELINED):	Depends on core variant
# INSTRUCTIONS:	11 TYPES
# ADDR MODES:	5 Load/Store
FP INSTRUCTIONS:	None
INTERRUPTS:	8 Exceptions, 2 External Interrupts (normal, fast)
VARIANTS:	ARM 2/250/3/6/60/600/610/620
	ARM 7/7m/7d/7dm/7di/7dmi/70/700/700i/710/71xxc/75xxfe/7tdmi
	ARM 8/810/9/9e/920/920t,,940t,9tdmi,
	StrongARM 110/1100
	Architectures: armv2, armv2a, armv3, armv3m, armv4, armv4t
	armv5, armv5t, armv5te
ATTRIBUTES:	Based on current independent studies, the ARM is the most popular RISC processor for embedded applications. Its architectural specification is highly scalable, and a number of variants have been defined and produced by several manufacturers. As a result, a wide range of configurations and performance levels exist, giving embedded designers many flexible design options. The simple instruction set architecture is easy to learn, yet powerful and optimized for compilers.
REFERENCES:	www.arm.com

The ARM Toolsuite consists of the following tools:

Tool Name	Tool Description	Version
arm-linux-gcc	GNU Compiler Collection (GCC)	3.4.4
arm-linux-g++	C++ compiler	3.4.4
arm-linux-as arm-linux-ld arm-linux-addr2line arm-linux-ar arm-linux-gcov arm-linux-nm arm-linux-objcopy arm-linux-objdump arm-linux-ranlib arm-linux-readelf arm-linux-size arm-linux-strings Arm-linux-strip	GNU assembler (as) GNU linker (ld) Converts addresses to file names & line # Creates object code archives Test coverage program for analysis Lists symbols from object files Copies and translates object files Displays information from object files Generates index to archive contents Displays information about ELF objects Lists file section sizes and total sizes Lists printable strings from files Strips debug symbols from binaries	2.15
arm-linux-gdb arm-linux-gdbtk arm-linux-run	GDB debugger Visual GDB Instruction Set Simulator	6.1
libc.a and libm.a	uClibc C and Math Libraries	0.9.26

TARGET CPU DEPENDENT INFORMATION

This toolsuite comprises a complete development environment for the ARM processor family. Included in this section of the Appendix are the required compiler and linker flags, a list of debugger targets for both command line `GDB` and `Visual GDB`, machine compiler and assembler dependent options, and notes.

Required Compiler / Linker Flags

The following compiler/linker options are required to build for execution with the simulator.

```
CFLAGS = (none required)
LFLAGS = (none required)
```

Compiler Machine Dependent Options

Section 3 describes all of the common compiler, linker, and assembler options used in Microcross GNU X-Tools Linux edition; in addition, these `` `-m' `` options are defined for ARM architectures:

Option	Description
-mapcs-frame	Generate a stack frame that is compliant with the ARM Procedure Call Standard for all functions, even if this is not strictly necessary for correct execution of the code. Specifying `'-fomit-frame-pointer'` with this option will cause the stack frames not to be generated for leaf functions. The default is `'-mno-apcs-frame'`.
-mapcs	This is a synonym for `'-mapcs-frame'`.
-mapcs-32	Generate code for a processor running with a 32-bit program counter, and conforming to the function calling standards for the APCS 32-bit option. This option replaces the `'-m6'` option of previous releases of the compiler and is now the default.
-mapcs-stack-check	Generate code to check the amount of stack space available upon entry to every function (that actually uses some stack space). If there is insufficient space available then either the function `'__rt_stkovf_split_small'` or `'__rt_stkovf_split_big'` will be called, depending upon the amount of stack space required. The run time system is required to provide these functions. The default is `'-mno-apcs-stack-check'`, since this produces smaller code. Must be used with `'-mapcs-frame'`.

Option	Description
-mno-sched-prolog	Prevent the reordering of instructions in the function prolog, or the merging of those instructions with the instructions in the function's body. This means that all functions will start with a recognizable set of instructions (or in fact one of a choice from a small set of different function prologues), and this information can be used to locate the start if functions inside an executable piece of code. The default is '-msched-prolog'.
-msoft-float	This is the default, and it is not necessary to specify it. The Instruction Set Simulator (ISS) will only run code generated with the default software floating point emulation. Software floating point is the default.
-mlittle-endian	Generate code for a processor running in little-endian mode. This is the default for all standard configurations. Big endian is not supported with this toolsuite.
-malignment-traps	Generate code that will not trap if the MMU has alignment traps enabled. On ARM architectures prior to ARMv4, there were no instructions to access half-word objects stored in memory. However, when reading from memory a feature of the ARM architecture allows a word load to be used, even if the address is unaligned, and the processor core will rotate the data as it is being loaded. This option tells the compiler that such misaligned accesses will cause a MMU trap and that it should instead synthesize the access as a series of byte accesses. The compiler can still use word accesses to load half-word data if it knows that the address is aligned to a word boundary. This option is ignored when compiling for ARM architecture 4 or later, since these processors have instructions to directly access half-word objects in memory.
-mno-alignment-traps	Generate code that assumes that the MMU will not trap unaligned accesses. This produces better code when the target instruction set does not have half-word memory operations (implementations prior to ARMv4). Note that you cannot use this option to access unaligned word objects, since the processor will only fetch one 32-bit aligned object from memory. The default setting for most targets is '-mno-alignment-traps', since this produces better code when there are no half-word memory instructions available.
-mcpu=<name>	This specifies the name of the target ARM processor. GCC uses this name to determine what kind of instructions it can use when generating assembly code. Permissible names are: arm2, arm250, arm3, arm6, arm60, arm600, arm610, arm620, arm7, arm7m (the default setting), arm7d, arm7dm, arm7di, arm7dmi, arm70, arm700, arm700i, arm710, arm710c, arm7100, arm7500, arm7500fe, arm7tdmi, arm8, strongarm, strongarm110, strongarm1100, arm8, arm810, arm9, arm920, arm920t, arm9tdmi, arm9e, and arm1020t, xscale.
-mtune=<name>	This option is very similar to the '-mcpu=' option, except that instead of specifying the actual target processor type, and hence restricting which instructions can be used, it specifies that GCC should tune the performance of the code as if the target were of the type specified in this option, but still choosing the instructions that it will generate based on the CPU specified by a '-mcpu=' option. For some arm implementations better performance can be obtained by using this option.
-march=<name>	This specifies the name of the target ARM architecture. GCC uses this name to determine what kind of instructions it can use when generating assembly code. This option can be used in conjunction with or instead of the '-mcpu=' option. Permissible names are: armv2, armv2a, armv3, armv3m, armv4 (the default setting), armv4t, armv5, and armv5te.
-mfpe=<number>	This specifies the version of the floating-point emulation available on

Option	Description
-mfp=<number>	the target. Permissible values are 2 and 3. '-mfp=' is a synonym for '-mfpe=' to support older versions of GCC.
-mstructure-size-boundary=<n>	The size of all structures and unions will be rounded up to a multiple of the number of bits set by this option. Permissible values are 8 and 32. The default value varies for different toolsuites. For the COFF targeted toolsuite the default value is 8. Specifying the larger number can produce faster, more efficient code, but can also increase the size of the program. The two values are potentially incompatible. Code compiled with one value cannot necessarily expect to work with code or libraries compiled with the other value, if they exchange information using structures or unions. Programmers are encouraged to use the 32 value as future versions of the toolsuite may default to this value.
-mabort-on-noreturn	Generate a call to the function abort at the end of a noreturn function. It will be executed if the function tries to return.
-mlong-calls -mno-long-calls	Tells the compiler to perform function calls by first loading the address of the function into a register and then performing a subroutine call on this register. This switch is needed if the target function will lie outside of the 64 megabyte addressing range of the offset based version of subroutine call instruction. Even if this switch is enabled, not all function calls will be turned into long calls. The heuristic is that static functions, functions which have the 'short-call' attribute, functions that are inside the scope of a '#pragma no_long_calls' directive and functions whose definitions have already been compiled within the current compilation unit, will not be turned into long calls. The exception to this rule is that weak function definitions, functions with the `long-call' attribute or the `section' attribute, and functions that are within the scope of a '#pragma long_calls' directive, will always be turned into long calls. This feature is not enabled by default. Specifying '--no-long-calls' will restore the default behavior, as will placing the function calls within the scope of a '#pragma long_calls_off' directive. Note these switches have no effect on how the compiler generates code to handle function calls via function pointers.
-msingle-pic-base	Treat the register used for PIC addressing as read-only, rather than loading it in the prologue for each function. The run-time system is responsible for initializing this register with an appropriate value before execution begins.
-mpic-register=<reg>	Specify the register to be used for PIC addressing. The default is r10 unless stack checking is enabled, when r9 is used. Must be used with –fpic option to be valid.

-mpoke-function-name	Write the name of each function into the text section, directly preceding the function prologue. The generated code is similar to this:
	t0
	.ascii "arm_poke_function_name", 0
	.align
	t1
	.word 0xff000000 + (t1 - t0)
	arm_poke_function_name
	mov ip, sp
	stmfd sp!, {fp, ip, lr, pc}
	sub fp, ip, #4
	When performing a stack backtrace, code can inspect the value of pc stored at fp + 0. If the trace function then looks at location pc - 12 and the top 8 bits are set, then we know that there is a function name embedded immediately preceding this location and has length ((pc[-3]) & 0xff000000).

Compiler-Defined Symbols and Standard Search Directories
From the command line in GNU X-Tools Shell, type:

```
$ xtools arm-linux (cr)
arm-linux$ gcc –v –E – (cr)
```

You will get all of the compiler-defined symbols printed out on the screen. Press `Ctrl-C` to exit this mode.

Assembler Machine Dependent Options
The following options are available when as is configured for the ARM processor family.

Options	Description

`-marm7[tdmi]`
`-marm8[10]`
`-marm9,920,9tdmi`
`-mstrongarm110[0]]`
`-marmv4 | -marmv4t`
`-marmv5 | -marmv5te`
	Arm 7 processors
	Arm 8 processors
	Arm 9 processors
	StrongARM processors
	Arm architectures

`-mfpe-old`	No float load/store multiples
`-mfpa10 | -mfpa11`	FPA10 and 11 co-processor
`-mapcs-32`	Select which procedure calling convention is in use: 32-bit APCS
`-EL`	Select little-endian (-EL) output – the default

GNU ARM Assembler Quick Reference

A summary of useful commands and expressions for the ARM architecture using the GNU assembler is presented briefly in the concluding portion of this Appendix. Each assembly line has the following format:

```
[<label>:]  [<instruction or directive>}  @ comment
```

Unlike the ARM assembler, using the GNU assembler does not require you to indent instructions and directives. Labels are recognized by the following colon instead of their position at the start of a line. An example follows showing a simple assembly program defining a function ‘`add`’ that returns the sum of two input arguments:

```
.section .text, "x"

.global     add                 @ give the symbol add external linkage

add:
      ADD   r0, r0, r1          @ add input arguments
      MOV   pc, lr              @ return from subroutine

                                @ end of program
```

GNU Assembler Directives for ARM

The follow is an alphabetical listing of the more command GNU assembler directives.

GNU Assembler Directive	Description
`.ascii "<string>"`	Inserts the string as data into the assembly (like DCB in armasm).
`.asciz "<string>"`	Like .ascii, but follows the string with a zero byte.

GNU Assembler Directive	Description
.balign <power_of_2> {,<fill_value> {,<max_padding>} }	Aligns the address to <power_of_2> bytes. The assembler aligns by adding bytes of value <fill_value> or a suitable default. The alignment will not occur if more than <max_padding> fill bytes are required (similar to ALIGN in armasm).
.byte <byte1> {,<byte2>} …	Inserts a list of byte values as data into the assembly (like DCB in armasm).
.code <number_of_bits>	Sets the instruction width in bits. Use 16 for Thumb and 32 for ARM assembly (similar to CODE16 and CODE32 in armasm).
.else	Use with .if and .endif (similar to ELSE in armasm).
.end	Marks the end of the assembly file (usually omitted).
.endif	Ends a conditional compilation code block – see .if, .ifdef, .ifndef (similar to ENDIF in armasm).
.endm	Ends a macro definition – see .macro (similar to MEND in armasm).
.endr	Ends a repeat loop – see .rept and .irp (similar to WEND in armasm).
.equ <symbol name>, <value>	This directive sets the value of a symbol (similar to EQU in armasm)
.err	Causes assembly to halt with an error.
.exitm	Exit a macro partway through – see .macro (similar to MEXIT in armasm)
.global <symbol>	This directive gives the symbol external linkage (similar to EXPORT in armasm).
.hword <short1> {,<short2>} …	Inserts a list of 16-bit values as data into the assembly (similar to DCW in armasm).
.if <logical_expression>	Makes a block of code conditional. End the block using .endif (similar to IF in armasm). See also .else.
.ifdef <symbol>	Include a block of code if <symbol> is defined. End the block with .endif.
.ifndef <symbol>	Include a block of code if <symbol> is not defined. End the block with .endif.
.include "<filename>"	Includes the indicated source file (similar to INCLUDE in armasm or #include in C).
.irp <param> {,<val_1>} {,<val_2>} …	Repeats a block of code, once for each value in the value list. Mark the end of the block using a .endr directive. In the repeated code block, use \<param> to substitute the associated value in the value list.

.macro <name> {<arg_1} {,<arg_2>} … {,<arg_N>}	Defines an assembler macro called <name> with N parameters. The macro definition must end with .endm. To escape from the macro at an earlier point, use .exitm. These directives are similar to MACRO, MEND, and MEXIT in armasm. You must precede the dummy macro parameters by \. For example:
	
	`.macro SHIFTLEFT a, b`
	`  .if \b < 0`
	`  MOV \a, \a, ASR #-\b`
	`  .exitm`
	`  .endif`
	`  MOV \a, \a, LSL #\b`
	`.endm`

.rept <number_of_times>	Repeats a block of code the given number of times. End with .endr.
<register_name> .req <register_name>	This directive names a register. It is similar to the RN directive in armasm except that you must supply a name rather than a number on the right (e.g., `acc .req r0`).
.section <section_name> {,"<flags>"}	Starts a new code or data section. Sections in GNU are called `.text`, a code section, `.data`, an initialized data section, and `.bss`, an uninitialized data section. These sections have default flags, and the linker understands the default names (similar directive to the armasm

GNU Assembler Directive	Description
	directive AREA). The following are allowable .section flags for ELF format files: <Flag> Meaning a allowable section w writable section x executable section
`.set <variable_name>, <variable_value>`	This directive sets the value of a variable. It is similar to SETA in armasm.
`.space <number_of_bytes> {,<fill_byte>}`	Reserves the given number of bytes. The bytes are filled with zero or <fill_byte> if specified (similar to SPACE in armasm).
`.word <word1> {,<word2>} …`	Inserts a list of 32-bit word values as data into the assembly (similar to DCD in armasm).

Assembler Special Characters / Syntax

Inline comment char:	`'@'`
Line comment char:	`'#'`
Statement separator:	`';'`
Immediate operand prefix:	`'#' or '$'`

Register Names

General registers:	`%r0 - %r15`	`($0 = const 0)`
FP registers:	`%f0 - %f7`	
Non-saved (temp) regs:	`%r0 - %r3, %r12`	
Saved registers:	`%r4 - %r10`	
Stack ptr register:	`%sp`	
Frame ptr register:	`%fp`	
Link (retn) register:	`%lr`	
Program counter:	`%ip`	
Status register:	`$psw`	
Status register flags:	`xPSR`	
(x = C current)	`xPSR_all`	
(x = S saved)	`xPSR_f`	
	`xPSR_x`	
	`xPSR_ctl`	
	`xPSR_fs`	
	`xPSR_fx`	
	`xPSR_fc`	
	`xPSR_cs`	
	`xPSR_cf`	
	`xPSR_cx`	
	`.. and so on`	

Arm Procedure Call Standard (APCS) Conventions

Argument registers:	`%a0 - %a4`	`(aliased to %r0 - %r4)`
Returned value regs:	`%v1 - %v6`	`(aliased to %r4 - %r9)`

Addressing Modes

`'rn'` in the following refers to any of the numbered registers, but not the control registers.

`addr`	Absolute addressing mode
`%rn`	Register direct
`[%rn]`	Register indirect or indexed
`[%rn,#n]`	Register based with offset
`#imm`	Immediate data

Machine Dependent Directives

`.arm`	Assemble using arm mode
`.thumb`	Assemble using thumb mode
`.code16`	Assemble using thumb mode
`.code32`	Assemble using arm mode
`.force_thumb`	Force thumb mode (even if not supported)
`.thumb_func`	Mark entry point as thumb coded (force bx entry)
`.ltorg`	Start a new literal pool

Opcodes
For detailed information on the machine instruction set, see this manual:

ARM Architecture Reference Manual, Addison-Wesley ISBN 0-201-73719-1

Here is a recommended book to get a lot of system developer information on the ARM architecture.

ARM System Developer's Guide, Morgan Kaufmann Publishers ISBN 1-55860-874-5 (alk.paper), authors: Andrew N. Sloss, Dominic Symes, Chris Wright, 2004

NOTES & ERRATA

Appendix 2. GNU X-Tools MIPS Linux Toolsuite

MODEL (alias):	MIPS (alias: mips-linux for Big Endian and mipsel-linux for Little Endian)
MFR:	MIPS, AMD, LSI Logic, IDT, Toshiba, NEC, et al
CPU TYPE:	32/64 LD/Store RISC Processor
GP REGS/SIZE:	32 x 32 Bit
ADDR SPACE:	32-Bit Flat (4 GB)
CYCLE RATE:	Various
APPROX INSTRUCTION THROUGHPUT (PIPELINED):	Pipelined, concurrent
# INSTRUCTIONS:	73
# ADDR MODES:	7
FP INSTRUCTIONS:	Yes: IEEE 754
VARIANTS:	Basic 32-bit version; includes support for: r2k r3k r4k r5k r6k r8k variants from multiple sources.
ATTRIBUTES:	Widely used in telecom, network applications. Very wide range of variants and multi-vendor support.
REFERENCES:	www.mips.com/publications/index.html

The MIPS Toolsuite consists of the following tools:

Tool Name	Tool Description	Version
`mips-linux-gcc`	GNU Compiler Collection (GCC)	3.40
`mips-linux-g++`	C++ compiler	3.40

`mips-linux-as`
`mips-linux-ld`
`mips-linux-addr2line`
`mips-linux-ar`
`mips-linux-gcov`
`mips-linux-nm`
`mips-linux-objcopy`
`mips-linux-objdump`
`mips-linux-ranlib`
`mips-linux-readelf`
`mips-linux-size`
`mips-linux-strings`
`mips-linux-strip`
	GNU assembler (as)
	GNU linker (ld)
	Converts addresses to file names & line #
	Creates object code archives
	Test coverage program for analysis
	Lists symbols from object files
	Copies and translates object files
	Displays information from object files
	Generates index to archive contents
	Displays information about ELF objects
	Lists file section sizes and total sizes
	Lists printable strings from files
	Strips debug symbols from binaries
		2.15

`mips-linux-gdb` `mips-linux-gdbtk`	GDB debugger Visual GDB	6.1
`libc.a and libm.a`	uClibc C and Math Libraries	0.9.26

TARGET CPU DEPENDENT INFORMATION

This toolsuite comprises a complete development environment for the MIPS32 processor family. Included in this section of the Appendix are the required compiler and linker flags, a list of debugger targets for both command line `GDB` and `Visual GDB`, machine compiler and assembler dependent options, and notes.

Required Compiler / Linker Flags
The following compiler/linker options are required to build for execution on the target.

```
CFLAGS = (none required)
LFLAGS = (none required)
```

Command Line GDB Debugger Targets

- `exec --` Allows loading and including source level browsing of the compiled executable within host memory.
- `remote --` Allows the `GDB` to connect to a remote target running the `GDB` remote debugging protocol via a serial port or a TCP socket connection. The `GDB` debugger can connect to a board via Abatron's BDI2000 or EPI's JEENI JTAG debug agents across Ethernet. Any debug agent that communicates using the `GDB` Stub protocol can be accessed through `GDB/Visual GDB`.
- `extended-remote --` Use a remote computer via a serial line.
- `async --` Use an executable file as a target.
- `extended-async --` Use a remote computer via a serial line.
- `ddb --` The `GDB` communicates with a remote target using the PMON MIPS remote debugging protocol via serial or socket port.
- `lsi --` The `GDB` communicates with a remote target using the mips PMON MIPS remote debugging protocol via serial.
- `mips --` The `GDB` communicates with a remote target using the MIPS remote debugging protocol via serial or socket port.
- `pmon --` The `GDB` communicates with a remote target using the mips PMON remote debugging protocol via serial or socket port.
- `sim --` The `GDB` debugger contains a integrated instruction set simulator based target that can load and execute compiled programs.

Visual GDB Debugger Targets

- Cisco/Serial
- Cisco/TCP
- DDB/Serial
- DDB/TCP
- MIPS/Serial
- MIPS/TCP
- PMON/Serial
- PMON/TCP
- Remote/Serial
- Remote/TCP
- Simulator
- GBDserver/Serial
- GDBserver/TCP

This toolsuite comprises a complete development environment for the mips-elf processor family. This toolsuite configuration has not been successfully tested with -mips3 (64-bit) code generated by the compiler. Recommend using Microcross' MIPS64 Toolsuite.

Compiler Machine Dependent Options
Section 3 describes all of the common compiler, linker, and assembler options used in Microcross GNU cross-tools; in addition, these `` `-m' `` options are defined for the MIPS family of computers:

Option	Description
`-march=`*arch*	Generate code that will run on arch, which can be the name of a generic MIPS ISA, or the name of a particular processor. The ISA names are: mips1, mips2, mips3, mips4, mips32, mips32r2, and mips64. The processor names are: 4kc, 4kp, 5kc, 20kc, m4k, r2000, r3000, r3900, r4000, r4400, r4600, r4650, r6000, r8000, rm7000, rm9000, orion, sb1, vr4100, vr4111, vr4120, vr4300, vr5000, vr5400 and vr5500. The special value from-abi selects the most compatible architecture for the selected ABI (that is, mips1 for 32-bit ABIs and mips3 for 64-bit ABIs). In processor names, a final 000 can be abbreviated as k (for example, -march=r2k). Prefixes are optional, and vr may be written r. GCC defines two macros based on the value of this option. The first is _MIPS_ARCH, which gives the name of target architecture, as a string. The second has the form _MIPS_ARCH_foo, where foo is the capitalized value of _MIPS_ARCH. For example, -march=r2000 will set _MIPS_ARCH to "r2000" and define the macro _MIPS_ARCH_R2000. Note that the _MIPS_ARCH macro uses the processor names given above. In other words, it will have the full prefix and will not abbreviate 000 as k. In the case of from-abi, the macro names the resolved architecture (either "mips1" or "mips3"). It names the default architecture when no -march option is given.
`-mtune=`*arch*	Optimize for arch. Among other things, this option controls the way instructions are scheduled, and the perceived cost of arithmetic operations. The list of arch values is the same as for -march. When this option is not used, GCC will optimize for the processor specified by -march. By using -march and -mtune together, it is possible to generate code that will run on a family of processors, but optimize the code for one particular member of that family. -mtune defines the macros _MIPS_TUNE and _MIPS_TUNE_foo, which work in the same way as the -march ones described above.
`-mips1`	Issue instructions from level 1 of the MIPS ISA. This is the default. `'r3000'` is the default CPU type at this ISA level.
`-mips2`	Issue instructions from level 2 of the MIPS ISA (branch likely, square root instructions). `'r6000'` is the default cpu type at this ISA level.
`-mips3`	Issue instructions from level 3 of the MIPS ISA (64-bit instructions). `'r4000'` is the default CPU type at this ISA level.
`-mips4`	Issue instructions from level 4 of the MIPS ISA (conditional move, prefetch, enhanced FPU instructions). `'r8000'` is the default CPU type at this ISA level.
`-mips32`	Equivalent to `-march=mips4`.
`-mips32r2`	Equivalent to `-march=mips32r2`.
`-mips64`	Equivalent to `-march=mips64`.
`-mips16` `-mno-mips16`	Enable/disable 16-bit instructions.
`-mabi=32`	Generate code for the indicated ABI.
`-mabicalls` `-mno-abicalls`	Emit (or do not emit) the pseudo operations `'.abicalls'`, `'.cpload'`, and `'.cprestore'` that some System V.4 ports use for position independent code.
`-mxgot` `-mno-xgot`	Lift (do not lift) the usual restrictions on the size of the global offset table. GCC normally uses a single instruction to load values from the GOT. While this is relatively efficient, it will only work if the GOT is

Option	Description
	smaller than about 64k. Anything larger will cause the linker to report an error such as: relocation truncated to fit: R_MIPS_GOT16 foobar If this happens, you should recompile your code with -mxgot. It should then work with very large GOTs, although it will also be less efficient, since it will take three instructions to fetch the value of a global symbol. Note that some linkers can create multiple GOTs. If you have such a linker, you should only need to use -mxgot when a single object file accesses more than 64k's worth of GOT entries. Very few do. These options have no effect unless GCC is generating position independent code.
`-membedded-pic` `-mno-embedded-pic`	Generate PIC code suitable for some embedded systems. All calls are made using PC relative address, and all data is addressed using the $gp register. No more than 65536 bytes of global data may be used. This requires GNU as and GNU ld, which do most of the work.
`-mgp32`	Assume that 32 32-bit general-purpose registers are available. This is the default.
`-mfp32`	Assume that 32 32-bit floating-point registers are available. This is the default.
`-mhard-float`	Generate output containing floating-point instructions. This is the default if you use the unmodified sources.
`-msoft-float`	Generate output containing library calls for floating point.
`-msingle-float` `-mdouble-float`	The `'-msingle-float'` switch tells GCC to assume that the floating-point coprocessor only supports single precision operations, as on the `'r4650'` chip. The `'-mdouble-float'` switch permits GCC to use double precision operations. This is the default.
`-mint64`	Force int and long types to be 64-bits wide. See `'-mlong32'` for an explanation of the default, and the width of pointers.
`-mlong64`	Force long types to be 64-bits wide. See `'-mlong32'` for an explanation of the default, and the width of pointers.
`-mlong32`	Force long, int, and pointer types to be 32-bits wide. If none of `'-mlong32'`, `'-mlong64'`, or `'-mint64'` is set, the size of ints, longs, and pointers depends on the ABI and ISA chosen. For `'-mabi=32'`, and `'-mabi=n32'`, ints and longs are 32-bits wide. For `'-mabi=64'`, ints are 32-bits, and longs are 64-bits wide. For `'-mabi=eabi'` and either `'-mips1'` or `'-mips2'`, ints and longs are 32-bits wide. For `'-mabi=eabi'` and higher ISAs, ints are 32-bits, and longs are 64-bits wide. The width of pointer types is the smaller of the width of longs or the width of general purpose registers (which in turn depends on the ISA).
`-G num`	Put global and static items less than or equal to num bytes into the small data or bss sections instead of the normal data or bss section. This allows the assembler to emit one word memory reference instructions based on the global pointer (gp or $28), instead of the normal two words used. By default, num is 8 when the MIPS assembler is used, and 0 when the GNU assembler is used. The `'-G num'` switch is also passed to the assembler and linker. All modules should be compiled with the same `'-G num'` value.

`-membedded-data`
 	
`-mno-embedded-data`
	Allocate variables to the read-only data section first if possible, then next in the small data section if possible, otherwise in data. This gives slightly slower code than the default, but reduces the amount of RAM required when executing, and thus may be

Option	Description
	preferred for some embedded systems.
-muninit-const-in-rodata -mno-uninit-const-in-rodata	When used together with -membedded-data, it will always store uninitialized const variables in the read-only data section.
-msplit-addresses -mno-split-addresses	Generate code to load the high and low parts of address constants separately. This allows GCC to optimize away redundant loads of the high order bits of addresses. This optimization requires GNU as and GNU ld. This optimization is enabled by default for some embedded targets where GNU as and GNU ld are standard.
-mexplicit-relocs -mno-explicit-relocs	Use (do not use) assembler relocation operators when dealing with symbolic addresses. The alternative, selected by -mno-explicit-relocs, is to use assembler macros instead. -mexplicit-relocs is usually the default if GCC was configured to use an assembler that supports relocation operators. However, there are two exceptions: GCC is not yet able to generate explicit relocations for the combination of -mabi=64 and -mno-abicalls. This will be addressed in a future release. The combination of -mabicalls and -fno-unit-at-a-time implies -mno-explicit-relocs unless explicitly overridden. This is because, when generating abicalls, the choice of relocation depends on whether a symbol is local or global. In some rare cases, GCC will not be able to decide this until the whole compilation unit has been read.
-mcheck-zero-division -mno-check-zero-division	Trap (do not trap) on integer division by zero. The default is -mcheck-zero-division.
-mmemcpy -mno-memcpy	The '-mmemcpy' switch makes all block moves call the appropriate string function ('memcpy' or 'bcopy') instead of possibly generating inline code.
-mlong-calls -mno-long-calls	Do all calls with the 'JALR' instruction, which requires loading up a function's address into a register before the call. You need to use this switch, if you call outside of the current 512-megabyte segment to functions that are not through pointers.
-mmad -mno-mad	Permit use of the `mad', `madu' and `mul' instructions, as on the 'r4650' chip.
-mfused-madd -mno-fused-madd	Enable (disable) use of the floating point multiply-accumulate instructions, when they are available. The default is -mfused-madd. When multiply-accumulate instructions are used, the intermediate product is calculated to infinite precision and is not subject to the FCSR Flush to Zero bit. This may be undesirable in some circumstances.
-nocpp	Tell the MIPS assembler to not run its preprocessor over user assembler files (with a '.s' suffix) when assembling them.
-mfix-sb1 -mno-fix-sb1	Work around certain SB-1 CPU core errata. (This flag currently works around the SB-1 revision 2 "F1" and "F2" floating point errata).
-mflush-func=func -mno-flush-func	Specifies the function to call to flush the I and D caches, or to not call any such function. If called, the function must take the same arguments as the common _flush_func(), that is, the address of the memory range for which the cache is being flushed, the size of the memory range, and the number 3 (to flush both caches). The default depends on the target GCC was configured for, but commonly is either _flush_func or __cpu_flush.
-mbranch-likely	Enable or disable use of Branch Likely instructions, regardless of the default for the selected architecture. By default, Branch Likely

Option	Description
-mno-branch-likely	instructions may be generated if they are supported by the selected architecture. An exception is for the MIPS32 and MIPS64 architectures and processors which implement those architectures; for those, Branch Likely instructions will not be generated by default because the MIPS32 and MIPS64 architectures specifically deprecate their use.

These options are defined by the macro 'TARGET_SWITCHES' in the machine description. The default for the options is also defined by that macro, which enables you to change the defaults.

Compiler-Defined Symbols
From the command line in GNU X-Tools Shell or xterm, type:

```
$ xtools mips-linux (cr)
mips-linux$ gcc –v –E – (cr)
```

You will get all of the compiler-defined symbols printed out on the screen. Press Ctrl-C to exit this mode.

Assembler Machine Dependent Options

Options	Description
-g, -g2	Do not remove unneeded NOPs or swap branches.
-G*NUM*	Allow referencing objects up to *NUM* bytes implicitly with the gp register [default 8].
-mips1	Generate MIPS ISA I instructions (r2000 and r3000 processors).
-mips2	Generate MIPS ISA II instructions (r6000 processor).
-mips3	Generate MIPS ISA III instructions (r4000 processor).
-mips4	Generate MIPS ISA IV instructions.
-mips5	Generate MIPS ISA V instructions.
-m4010	Generate code for LSI R4010 using addciu, addffc, etc.
-no-m4010	Negates LSI R4010 mode.
-nocpp	Ignored, accepted for compatibility /w other MIPS assemblers.
-no-m*CPU* -m*CPU*	Don't generate code specific to *CPU*. For -m*CPU* and -no-m*CPU*, *CPU* must be one of: 3900, 4010, 4100.
-O0 (two capital O's)	Remove unneeded NOPs, do not swap branches.
-O (single capital O)	Remove unneeded NOPs and swap branches.
-trap, -no-break	Trap exception on div by 0 and mult overflow. Not supported for ISA 1 (-mips2 and higher).
-break, -no-trap	Break exception on div by 0 and mult overflow.
-KPIC, -call_shared	Generate SVR4 position independent code.
-non_shared	Do not generate position independent code.
-xgot	Assume a 32-bit GOT.
-32	Create 32-bit object file (default).

Assembler Special Characters/Syntax

Inline comment char: '#'
Line comment char: '#'
Statement separator: none
Other notes: '$' used as register designator

Register Names

General registers: $0-$31
FP registers: $f0-$f31
Value registers: $v0,$v1
Argument registers: $a0-$a3
Temporary registers: $t0-$t7

Saved registers:	`$s0-$s7`
Temp expr registers:	`$t8,$t9`
Kernel resvd registers:	`$kt0,$kt1`
Global ptr register:	`$gp`
Stack ptr register:	`$sp`
Frame ptr register:	`$fp`
Link (retn) register:	`$ra`
FP result registers:	`$fv0,$fv1`
FP temp registers:	`$ft1-$ft3`
FP working registers:	`$ft0-$ft7`
FP argument registers:	`$fa0-$fa7`
FP expr registers:	`$ft8-$ft11`
FP saved registers:	`$fs0-$fs7`
Program counter:	`$pc`
Status register:	`$psr`

Addressing Modes

'`rn`' in the following refers to any of the numbered registers, but not the control registers.

`addr`	Absolute addressing mode
`$rn`	Register direct
`($rn)`	Register indirect (also called register based)
`expr($rn)`	Register based with offset
`$rx($rn)`	Register based with register offset
`imm`	Immediate data

Machine Dependent Directives

`.set mipsn`	Changes MIPS architecture level during assembly: n=1,2,3,4
`.set mips0`	Reverts MIPS architecture level to invocation value
`.set mips16`	Changes to MIPS 16 mode. Assembles MIPS 16 instructions
`.set nomips16`	Revert to normal 32-bit mode
`.insn`	Marks code emitted by data directive as an instruction (sets lsb) Useful for MIPS16 code segments
`.rodata, .sbss`	

MIPS supports some additional sections besides the usual '`.text`', '`.data`' and '`.bss`'. The additional sections are '`.rodata`', used for read-only data, '`.sdata`', used for small data, and '`.sbss`', used for small common objects.

The assembler uses the '`$gp`' ($28) register to form the address of a small object. Any object in the '`.sdata`' or '`.sbss`' sections is considered small in this sense. For external objects, or for objects in the '`.bss`' section, you can use the `GCC '-G'` option to control the size of objects addressed via '`$gp`'; the default value is 8, meaning that a reference to any object eight bytes or smaller uses '`$gp`'. Passing '`-G 0`' to as prevents it from using the '`$gp`' register on the basis of object size (but the assembler uses '`$gp`' for objects in '`.sdata`' or '`.sbss`' in any case). The size of an object in the '`.bss`' section is set by the '`.comm`' or '`.lcomm`' directive that defines it. The size of an external object may be set with the .extern directive. For example, '`.extern sym,4`' declares that the object at sym is 4 bytes in length, while leaving sym otherwise undefined.

Using small objects requires linker support, and assumes that the '`$gp`' register is correctly initialized (normally done automatically by the startup code). MIPS assembly code must not modify the '`$gp`' register.

Opcodes

For detailed information on the machine instruction set, see these manuals:

MIPS32 Architecture for Programmers Vol1: *Introduction to the MIPS32 Architecture*
MIPS Technologies Document No MD00082

MIPS32 Architecture for Programmers Vol2: *MIPS32 Instruction Set*
MIPS Technologies Document No MD00086

MIPS32 Architecture for Programmers Vol3: *MIPS32 Privileged Resource Architecture*
MIPS Technologies Document No MD00090

NOTES & ERRATA

Appendix 3. GNU X-Tools PowerPC Linux Toolsuite

MODEL (target alias):	PowerPC (alias: ppc-linux)
MFR:	Freescale and AMCC (formerly IBM's product line)
CPU TYPE:	32-Bit RISC Load/Store
GP REGS/SIZE:	32 x 32-Bit
ADDR SPACE:	32-Bit (4 GB Flat or VM)
APPROX. INSTRUCTION THROUGHPUT (PIPELINED):	Up to 4 instructions per cycle (some variants) branch prediction.
FP INSTRUCTIONS:	IEEE 754 FPU (some variants)
VARIANTS:	Variants include: 4xx 5xx 6xx/e 7xx 74xx 8xx 9xx Rios Rios1 Rios2 Rsc G3, G4, G5
ATTRIBUTES:	This toolsuite comprises a complete development environment for the PowerPC processor family. It is widely supported, has a scalable architecture, and has many on-chip peripherals available.
REFERENCES:	www.freescale.com

The PowerPC Toolsuite consists of the following tools:

Tool Name	Tool Description	Version
`ppc-linux-gcc`	GNU Compiler Collection (GCC)	3.40
`ppc-linux-g++`	C++ compiler	3.40

`ppc-linux-as`
`ppc-linux-ld`
`ppc-linux-addr2line`
`ppc-linux-ar`
`ppc-linux-gcov`
`ppc-linux-nm`
`ppc-linux-objcopy`
`ppc-linux-objdump`
`ppc-linux-ranlib`
`ppc-linux-readelf`
`ppc-linux-size`
`ppc-linux-strings`
`ppc-linux-strip`
	GNU assembler (as)
	GNU linker (ld)
	Converts addresses to file names & line #
	Creates object code archives
	Test coverage program for analysis
	Lists symbols from object files
	Copies and translates object files
	Displays information from object files
	Generates index to archive contents
	Displays information about ELF objects
	Lists file section sizes and total sizes
	Lists printable strings from files
	Strips debug symbols from binaries
		2.15

`ppc-linux-gdb` `ppc-linux-gdbtk`	GDB debugger Visual GDB	6.1
`libc.a and libm.a`	uClibc C and Math Libraries	0.9.26

TARGET CPU DEPENDENT INFORMATION

This toolsuite comprises a complete development environment for the PowerPC processor family. Included in this section of the Appendix are the required compiler and linker flags, a list of debugger targets for both command line `GDB` and `Visual GDB`, machine compiler and assembler dependent options, and notes.

Required Compiler / Linker Flags

The following compiler/linker options are required to build for execution on the target.

```
CFLAGS = none
LFLAGS = none
```

Command Line GDB Debugger Targets

- `async --` Use a remote computer via a serial line.
- `Cisco --` Use a remote machine via TCP.
- `dink32 --` The `GDB` debugger communicates with a remote target processor running the DINK32 monitor.
- `exec --` Allows loading and including source level browsing of the compiled executable within host memory.
- `extended-async --` Use a remote computer via a serial line.
- `extended-remote --` Use a remote computer via a serial line.
- `remote --` Allows the `GDB` to connect to a remote target running the `GDB` remote debugging protocol via a serial port or a TCP socket connection. The `GDB` debugger can connect to a board via Abatron's BDI2000 or EPI's JEENI JTAG debug agents across Ethernet. Any debug agent that communicates using the `GDB` Stub protocol can be accessed through `GDB/Visual GDB`.
- `ppcbug --` The `GDB` debugger communicates with a remote target processor running the PowerPC PPCBug monitor on port 0.
- `ppcbug1 --` The `GDB` debugger communicates with a remote target processor running the PowerPC PPCBug monitor on port 1.
- `sds --` The `GDB` debugger communicates with a remote target processor running the SDS debugging protocol.
- `sim --` The `GDB` debugger contains a integrated simulator based target that can load and execute compiled programs.

Visual GDB Debugger Targets

- Cisco/Serial
- Cisco/TCP
- Remote/Serial
- Remote/TCP
- SDS
- Simulator
- GDBserver/Serial
- GDBserver/TCP

Compiler Machine Dependent Options

Section 3 describes all of the common compiler, linker, and assembler options used in Microcross GNU cross-tools; in addition, these `` `-m' `` options are defined for the IBM RS/6000 and PowerPC:

Option	Description
`-mcpu=`*`cpu_type`*	Set architecture type, register usage, choice of mnemonics, and instruction scheduling parameters for machine type cpu_type. Supported values for *cpu_type* are `401, 403, 405, 405fp, 505, 601, 602, 603, 603e, 604, 604e, 620, 630, 740, 7400, 7450, 750, 801, 821, 823, 860, 970, common, ec603e, G3, G4, G5, power, power2, power3,`

Option	Description
	`power4, powerpc, powerpc64, rios, rios1, rios2, rsc, and rs64a.` Other CPU types may be derivatives of one of the above core architectures. `-mcpu=common` selects a completely generic processor. Code generated under this option will run on any POWER or PowerPC processor. GCC will use only the instructions in the common subset of both architectures, and will not use the MQ register. GCC assumes a generic processor model for scheduling purposes. `-mcpu=power, -mcpu=power2, -mcpu=powerpc, and -mcpu=powerpc64` specify generic POWER, POWER2, pure 32-bit PowerPC (i.e., not MPC601), and 64-bit PowerPC architecture machine types, with an appropriate, generic processor model assumed for scheduling purposes. The other options specify a specific processor. Code generated under those options will run best on that processor, and may not run at all on others. The `-mcpu` options automatically enable or disable the following options: `-maltivec, -mhard-float, -mmfcrf, -mmultiple, -mnew-mnemonics, -mpower, -mpower2, -mpowerpc64, -mpowerpc-gpopt, -mpowerpc-gfxopt, -mstring.` The particular options set for any particular CPU will vary between compiler versions, depending on what setting seems to produce optimal code for that CPU; it doesn't necessarily reflect the actual hardware's capabilities. If you wish to set an individual option to a particular value, you may specify it after the `-mcpu` option, like `-mcpu=970 -mno-altivec.` On AIX, the `-maltivec` and `-mpowerpc64` options are not enabled or disabled by the `-mcpu` option at present, since AIX does not have full support for these options. You may still enable or disable them individually if you're sure it'll work in your environment.
`-mtune=`*`cpu_type`*	Set the instruction scheduling parameters for machine type *`cpu_type`*, but do not set the architecture type, register usage, or choice of mnemonics, as `-mcpu=`*`cpu_type`* would. The same values for cpu_type are used for `-mtune` as for `-mcpu`. If both are specified, the code generated will use the architecture, registers, and mnemonics set by `-mcpu`, but the scheduling parameters set by `-mtune.`
`-maltivec` `-mno-altivec`	These switches enable or disable the use of built-in functions that allow access to the AltiVec instruction set. You may also need to set `-mabi=altivec` to adjust the

Option	Description
	current ABI with AltiVec ABI enhancements.
-mabi=spe	Extend the current ABI with SPE ABI extensions. This does not change the default ABI, instead it adds the SPE ABI extensions to the current ABI.
-mabi=no-spe	Disable Booke SPE ABI extensions for the current ABI.
-misel=yes/no	This switch enables or disables the generation of ISEL instructions.
-mspe=yes/no	This switch enables or disables the generation of SPE simd instructions.
-mfloat-gprs=yes/no	This switch enables or disables the generation of floating point operations on the general purpose registers for architectures that support it. This option is currently only available on the MPC8540.
-mfull-toc -mno-fp-in-toc -mno-sum-in-toc -mminimal-toc	Modify generation of the TOC (Table Of Contents), which is created for every executable file. The '-mfull-toc' option is selected by default. In that case, GCC will allocate at least one TOC entry for each unique non-automatic variable reference in your program. GCC will also place floating-point constants in the TOC. However, only 16,384 entries are available in the TOC. If you receive a linker error message saying you have overflowed the available TOC space, then you can reduce the amount of TOC space used with the '-mno-fp-in-toc' and '-mno-sum-in-toc' options. '-mno-fp-in-toc' prevents GCC from putting floating-point constants in the TOC and '-mno-sum-in-toc' forces GCC to generate code to calculate the sum of an address and a constant at run-time instead of putting that sum into the TOC. You may specify one or both of these options. Each causes GCC to produce very slightly slower and larger code at the expense of conserving TOC space. If you still run out of space in the TOC even when you specify both of these options, specify '-mminimal-toc' instead. This option causes GCC to make only one TOC entry for every file. When you specify this option, GCC will produce code that is slower and larger but which uses extremely little TOC space. You may wish to use this option only on files that contain less frequently executed code.
-malign-natural -malign-power	On AIX, Darwin, and 64-bit PowerPC GNU/Linux, the option -malign-natural overrides the ABI-defined alignment of larger types, such as floating-point doubles, on their natural size-based boundary. The option -malign-power instructs GCC to follow the ABI-specified alignment rules. GCC defaults to the standard alignment defined in the ABI.
-msoft-float -mhard-float	Generate code that does not use (uses) the floating-point register set. Software floating-point emulation is provided if you use the '-msoft-float' option, and pass the option to GCC when linking. The default setting is hardware floating point.
-mmultiple -mno-multiple	Generate code that uses (does not use) the load multiple word instructions and the store multiple word instructions. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use '-mmultiple' on little endian PowerPC systems, since those instructions do not work when the processor is in little endian mode. The exceptions are PPC740 and PPC750, which permit the instructions usage in little endian mode.
-mstring -mno-string	Generate code that uses (does not use) the load string instructions and the store string word instructions to save multiple

Option	Description
	registers and do small block moves. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. **Warning:** Do not use `'-mstring'` on little endian PowerPC systems, since those instructions do not work when the processor is in little endian mode. The exceptions are PPC740 and PPC750, which permit the instructions usage in little endian mode.
`-mupdate` `-mno-update`	Generate code that uses (does not use) the load or store instructions that update the base register to the address of the calculated memory location. These instructions are generated by default. If you use `'-mno-update'`, there is a small window between the time that the stack pointer is updated and the address of the previous frame is stored, which means code that walks the stack frame across interrupts or signals may get corrupted data.
`-mfused-madd` `-mno-fused-madd`	Generate code that uses (does not use) the floating point multiply and accumulate instructions. These instructions are generated by default if hardware floating is used.
`-mno-bit-align` `-mbit-align`	On System V.4 and embedded PowerPC systems do not (do) force structures and unions that contain bit fields to be aligned to the base type of the bit field. For example, by default a structure containing nothing but 8 unsigned bit fields of length 1 would be aligned to a 4 byte boundary and have a size of 4 bytes. By using `'-mno-bit-align'`, the structure would be aligned to a 1 byte boundary and be one byte in size.
`-mno-strict-align` `-mstrict-align`	On System V.4 and embedded PowerPC systems do not (do) assume that unaligned memory references will be handled by the system.
`-mrelocatable` `-mno-relocatable`	On embedded PowerPC systems generate code that allows (does not allow) the program to be relocated to a different address at runtime. If you use `'-mrelocatable'` on any module, all objects linked together must be compiled with `'-mrelocatable'` or `'-mrelocatable-lib'`.
`-mrelocatable-lib` `-mno-relocatable-lib`	On embedded PowerPC systems generate code that allows (does not allow) the program to be relocated to a different address at runtime. Modules compiled with `'-mrelocatable-lib'` can be linked with either modules compiled without `'-mrelocatable'` and `'-mrelocatable-lib'` or with modules compiled with the `'-mrelocatable'` options.
`-mno-toc` `-mtoc`	On System V.4 and embedded PowerPC systems do not (do) assume that register 2 contains a pointer to a global area pointing to the addresses used in the program.
`-mlittle` `-mlittle-endian`	On System V.4 and embedded PowerPC systems compile code for the processor in little endian mode. The `'-mlittle-endian'` option is the same as `'-mlittle'`.
`-mbig` `-mbig-endian`	On System V.4 and embedded PowerPC systems compile code for the processor in big endian mode. The `'-mbig-endian'` option is the same as `'-mbig'`.
`-mprioritize-restricted-insns=priority`	This option controls the priority that is assigned to dispatch-slot restricted instructions during the second scheduling pass. The argument priority takes the value 0/1/2 to assign no/highest/second-highest priority to dispatch slot restricted instructions.
`-msched-costly-dep=dependence_type`	This option controls which dependences are considered costly by the target during instruction scheduling. The argument dependence_type takes one of the following

Option	Description
	values: no: no dependence is costly, `all`: all dependences are costly, `true_store_to_load`: a true dependence from store to load is costly, `store_to_load`: any dependence from store to load is costly, `number`: any dependence which latency >= `number` is costly.
`-minsert-sched-nops=scheme`	This option controls which nop insertion scheme will be used during the second scheduling pass. The argument scheme takes one of the following values: `no`: Don't insert nops. `pad`: Pad with nops any dispatch group which has vacant issue slots, according to the scheduler's grouping. `regroup_exact`: Insert nops to force costly dependent insns into separate groups. Insert exactly as many nops as needed to force an insn to a new group, according to the estimated processor grouping. `number`: Insert nops to force costly dependent insns into separate groups. Insert *number* nops to force an insn to a new group.
`-mcall-sysv`	On System V.4 and embedded PowerPC systems compile code using calling conventions that adheres to the March 1995 draft of the System V Application Binary Interface, PowerPC processor supplement. This is the default unless you configured GCC using `'powerpc-*-eabiaix'`.
`-mcall-sysv-eabi`	Specify both `'-mcall-sysv'` and `'-meabi'` options.
`-mcall-sysv-noeabi`	Specify both `'-mcall-sysv'` and `'-mno-eabi'` options.
`-mcall-linux`	On System V.4 and embedded PowerPC systems compile code for the Linux-based GNU system.
`-mcall-gnu`	On System V.4 and embedded PowerPC systems compile code for the Hurd-based GNU system.
`-mcall-netbsd`	On System V.4 and embedded PowerPC systems compile code for the NetBSD operating system.
`-maix-struct-return`	Return all structures in memory (as specified by the AIX ABI).
`-mabi=altivec`	Extend the current ABI with AltiVec ABI extensions. This does not change the default ABI, instead it adds the AltiVec ABI extensions to the current ABI.
`-mabi=no-altivec`	Disable AltiVec ABI extensions for the current ABI.
`-mprototype` `-mno-prototype`	On System V.4 and embedded PowerPC systems assume that all calls to variable argument functions are properly prototyped. Otherwise, the compiler must insert an instruction before every non prototyped call to set or clear bit 6 of the condition code register (CR) to indicate whether floating point values were passed in the floating point registers in case the function takes a variable arguments. With `'-mprototype'`, only calls to prototyped variable argument functions will set or clear the bit.
`-msim`	On embedded PowerPC systems, assume that the startup module is called `'sim-crt0.o'` and that the standard C libraries are `'libsim.a'` and `'libc.a'`. This is the default for `'powerpc-*-eabisim'`. configurations.
`-mmvme`	On embedded PowerPC systems, assume that the startup module is called `'crt0.o'` and the standard C libraries are `'libmvme.a'` and `'libc.a'`.
`-mads`	On embedded PowerPC systems, assume that the startup module is called `'crt0.o'` and the standard C libraries are `'libads.a'` and `'libc.a'`.
`-myellowknife`	On embedded PowerPC systems, assume that the startup module is called `'crt0.o'` and the standard C libraries are `'libyk.a'` and `'libc.a'`.

Option	Description
-mwindiss	Specify that you are compiling for the WindISS simulation environment.
-memb	On embedded PowerPC systems, set the PPC_EMB bit in the ELF flags header to indicate that 'eabi' extended relocations are used.
-meabi -mno-eabi	On System V.4 and embedded PowerPC systems do (do not) adhere to the Embedded Applications Binary Interface (eabi) that is a set of modifications to the System V.4 specifications. Selecting -meabi means that the stack is aligned to an 8 byte boundary, a function '_eabi' is called to from main to set up the eabi environment, and the '-msdata' option can use both r2 and r13 to point to two separate small data areas. Selecting '-mno-eabi' means that the stack is aligned to a 16-byte boundary, do not call an initialization function from main, and the '-msdata' option will only use r13 to point to a single small data area. The '-meabi' option is on by default if you configured GCC using one of the 'powerpc*-*-eabi*' options.
-msdata=eabi	On System V.4 and embedded PowerPC systems, put small initialized const global and static data in the '.sdata2' section, which is pointed to by register r2. Put small initialized non-const global and static data in the '.sdata' section, which is pointed to by register r13. Put small-uninitialized global and static data in the '.sbss' section, which is adjacent to the '.sdata' section. The '-msdata=eabi' option is incompatible with the '-mrelocatable' option. The '-msdata=eabi' option also sets the '-memb' option.
-msdata=sysv	On System V.4 and embedded PowerPC systems, put small global and static data in the '.sdata' section, which is pointed to by register r13. Put small-uninitialized global and static data in the '.sbss' section, which is adjacent to the '.sdata' section. The '-msdata=sysv' option is incompatible with the '-mrelocatable' option.
-msdata=default -msdata	On System V.4 and embedded PowerPC systems, if '-meabi' is used, compile code the same as '-msdata=eabi', otherwise compile code the same as '-msdata=sysv'.
-msdata=none -mno-sdata	On embedded PowerPC systems, put all initialized global and static data in the '.data' section, and all uninitialized data in the '.bss' section.
-G num	On embedded PowerPC systems, put global and static items less than or equal to num bytes into the small data or bss sections instead of the normal data or bss section. By default, num is 8. The '-G num' switch is also passed to the linker. All modules should be compiled with the same '-G num' value.
-mregnames -mno-regnames	On System V.4 and embedded PowerPC systems do (do not) emit register names in the assembly language output using symbolic forms.
-mlongcall -mno-longcall	Default to making all function calls via pointers, so that functions which reside further than 64 megabytes (67,108,864 bytes) from the current location can be called. This setting can be overridden by the shortcall function attribute, or by #pragma longcall(0). Some linkers are capable of detecting out-of-range calls and generating glue code on the fly. On these systems, long calls are unnecessary and generate slower code. As of this writing, the AIX linker can do this, as can the GNU linker for PowerPC/64. It is planned to add this feature to

Option	Description
	the GNU linker for 32-bit PowerPC systems as well. On Mach-O (Darwin) systems, this option directs the compiler emit to the glue for every direct call, and the Darwin linker decides whether to use or discard it. In the future, we may cause GCC to ignore all longcall specifications when the linker is known to generate glue.
`-pthread`	Adds support for multithreading with the pthreads library. This option sets flags for both the preprocessor and linker.

Compiler-Defined Symbols
From the command line in GNU X-Tools Shell, type:

```
$ xtools ppc-linux (cr)
ppc-linux$ gcc –v –E – (cr)
```

You will get all of the compiler-defined symbols printed out on the screen. Press `Ctrl-C` to exit this mode.

Assembler Machine Dependent Options

Options	Description
`-u`	Ignored*
`-mpwrx, -mpwr2`	Generate code for IBM POWER/2 (RIOS2).
`-mpwr`	Generate code for IBM POWER (RIOS1).
`-m601`	Generate code for Motorola PowerPC 601.
`-mppc, -mppc32, -m403, -m603, -m604`	Generate code for Motorola PowerPC 603/604.
`-mppc64, -m620`	Generate code for Motorola PowerPC 620.
`-mppc64bridge`	Generate code for PowerPC 64, including bridge insns.
`-mcom`	Generate code Power/PowerPC common instructions.
`-many`	Generate code for any architecture (PWR/PWRX/PPC).
`-mregnames`	Allow symbolic names for registers.
`-mno-regnames`	Do not allow symbolic names for registers.
`-mrelocatable`	Support for `GCC's -mrelocatble` option.
`-mrelocatable-lib`	Support for `GCC's -mrelocatble-lib` option.
`-memb`	Set `PPC_EMB` bit in ELF flags.
`-mlittle, -mlittle-endian`	Generate code for a little endian machine.
`-mbig, -mbig-endian`	Generate code for a big endian machine.
`-msolaris`	Generate code for Solaris.
`-mno-solaris`	Do not generate code for Solaris.
`-V`	Print assembler version number.
`-Qy, -Qn`	Ignored*

* Option is preserved for backwards compatibility with older makefiles and tools.

Assembler Special Characters/Syntax

Inline comment char: `'#' '!'`
Line comment char: `'#'`
Statement separator: `';'`

Register Names

General registers: `%r0 - %r31`

FP registers:	%f0 - %f31
Stack ptr register:	%sp (alias for %r1)
Frame ptr register:	%fp
Link (retn) register:	%lr
Program counter:	%pc
Status register:	%cr0 - %cr7
Count register:	%ctr
Small data pointer:	%r13

C Calling Conventions

Parameter:	p1 - p8	%r3 - %r10
Return value:	v1:v2	%r3:%r4
FP parameters:	p1 - p8	%fp1 - %fp8
FP return value:	v1	%fp1
Saved regs:		%r14 - %r31
		%f14 - %f31

Addressing Modes

'rn' in the following refers to any of the numbered registers, but not the control registers. The memory layout is big-endian.

addr	Absolute addressing mode
rn	Register direct
(rn)	Register indirect (also called register based)
(off,rn)	Register based with offset
	Register based with register offset
value	Immediate data (context sensitive, most instructions end in 'i')

Machine Dependent Directives

.rodata	Allocate data to read-only data section

Opcodes

For detailed information on the machine instruction set, see these manuals:

The PowerPC™ Architecture: *A Specification for a New Family of RISC processors*
Morgan Kaufmann Publishing

PowerPC™ Microprocessor Family: *The Programming Environments*
Motorola Doc number MPCFPE/AD

NOTES & ERRATA

Appendix 4. Common Minicom Problems

If you are having problems with Minicom, first try clearing all settings for Modem and dialing under Configure Minicom. Save the settings; this requires root permissions.

Problem	Solution
Error message: *Device /dev/ttyS0 access failed: Permission denied.*	Permissions for device are set improperly. As root user, enter # chmod o+rw /dev/ttyS0
No terminal activity.	• In Configure Minicom --> Serial port setup, check that Serial Device is consistent with the port you are using. • In Configure Minicom --> Serial port setup, ensure that Hardware Flow Control and Software Flow Control are both set to No. • If possible, connect the serial cable to another serial port on a Linux workstation and verify data can be exchanged over the serial cable in both directions.
Settings revert to original when Minicom is restarted.	Make sure you use Configure Minicom --> Save setup as dfl after making changes. To do this, you must run Minicom as the root user.

Appendix 5. Glossary

TERM	Description
ARM Ltd	ARM Ltd is the developer of the ARM embedded RISC microprocessor family.
ARP	Address resolution protocol.
Board Support Package	Tools, drivers, Linux kernel, and documentation necessary to support development on a particular board.
Bootloader	Lucent Technologies, Inc.'s code used to load code to flash and set boot options. See uMON documentation.
BOOTP	Network protocol allowing a network device to obtain an IP address and other information from a server.
bpp	Bits per pixel.
BSP	See "Board support package."
BusyBox	Open-source code that combines tiny versions of common utilities and a shell into a single small executable.
CLUT	Color look-up table
Cross-build	The process of running a toolchain on your host computer to produce executables for a different target file system, also known as "cross-compile." For example, you may cross-build on the host to produce files for the real target hardware.
Daemon	A process that typically runs in the background, behind the scenes, that is usually invoked as part of the system startup scripts. These processes usually run continuously as helper utilities for various system operations such as networking.
DHCP	Dynamic Host Configuration Protocol server.
Digital Signal Processor	A microprocessor specialized to process data in real-time. Certain operations can be executed in parallel, which results in higher processing capacity for a given clock speed.
DSP	See "Digital Signal Processor."
ELF	ELF (Executable and Linking Format) is a binary format originally developed by USL (UNIX System Laboratories). Because of its increased flexibility over the older a.out format that Linux previously used, the GCC and C library developers decided to move to using ELF as the Linux standard binary format also.
File system	The directory structure (beginning with "/") comprising configuration files, applications, libraries, and an operating system.
GDB	Gnu DeBugger. Allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.
GIO or GPIO	General Purpose Input / Output hardware lines.
General Purpose Processor	A microprocessor that is not specialized for real-time parallel processing of data.
GNOME	See "GNU Networked Object Model Environment."
GNU Networked Object Module Environment	A user-friendly set of applications and desktop tools used in conjunction with a window manager for the X Window System. GNOME is similar in purpose and scope to CDE and KDE, but GNOME is based completely on Open Source software.
GPP	See "General purpose processor."
GUI	Graphical user interface.
Host Computer	The Linux-based desktop system used for software development.
Host Linux	Refers to the Linux operating system running on the host computer. See also "User-mode Linux."

TERM	Description
Host Shell	The Linux terminal shell running on the host computer. The default prompt shell is "$" for normal users and "#" for root.
IP	Internet Protocol.
JFFS	The Journaling Flash File System, developed by Axis Communications in Sweden, is aimed at providing a crash/power-down-safe file system for diskless embedded devices. It is released under the GPL, and the current version works for the Linux 2.0 kernel series and memory-mapped industry-standard flash-memories (aka "NOR-flashes").
JFFS2	A Journaling Flash File System based on the original JFFS and developed by Red Hat. JFFS2 offers improved performance, compression, RAM footprint, concurrency and support for suspending flash erases, and the support for hard links.
JTAG	Joint Test Action Group, specifies test framework for electronic logic components; IEEE Standard 1149.1, also known as IEEE 1149.1 (JTAG) or Boundary Scan.
KDE	See "K Desktop Environment."
K Desktop Environment	A powerful Open Source graphical desktop environment for Unix workstations.
Linux Kernel	The core of the Linux operating system, including networking.
Linux Workstation	The computer running the Linux operating system that is used for developing, compiling, and building the kernel and target file system that is loaded and executed on the SBC.
Minicom	Minicom is a menu-driven, serial communication program, and includes ANSI color, dialing directory, dial-a-list, script language, and so forth. Minicom is a clone of the MS-DOS Telix program.
MMU	The Memory Management Unit (MMU) is a component of a processor architecture. The MMU is responsible for mapping physical addresses defined by the hardware configuration into the virtual addresses the are used by the Linux-based software. All Linux kernels require that the processor support an MMU; uClinux is a special version of Linux for processors that do not support an MMU.
NFS	The Network File System (NFS) allows machines to mount a disk partition on a remote machine as if it were on a local hard drive. This allows for fast, seamless sharing of files across a network.
OSD	On-Screen Display.
SBC	Single Board Computer.
SDK	Software Development Kit.
SOC	System-On-a-Chip.
Target File System	The file system for a particular target hardware.
Target Hardware	The hardware running Linux.
TFS	Trivial File System. This is the small flash file system managed by the uMON bootloader.
TI	Texas Instruments, Inc.
Toolchain	Tools and utilities used to build applications and other code for the target hardware.
UI	User Interface.
uMON	MicroMonitor. The Lucent Technologies, Inc. bootloader allows users to manage the loading, storing, and invoking of a Linux kernel and root file system. In normal operation, the bootloader resides in flash and is the first program run when powering up. Unless intercepted, uMON will typically transfer control to the stored system.
x86	Any processor compatible with the Intel 386, 486, or Pentium family.
XIP	Execute In Place.

Appendix 6. Additional Resources

ARM, Ltd
www.arm.com

Atmel Corporation
www.atmel.com

BusyBox, the Swiss Army Knife for Embedded Linux
www.busybox.net

Cogent Computer Systems, Incorporated
www.cogcomp.com

Linux documentation, including guides and FAQs
www.linuxdoc.org

Linux newbie.org for new Linux users
www.linuxnewbie.org

Linux kernel organization that controls new releases and has archives of old ones
www.kernel.org

Memory Technology Device (MTD) for details on memory (Flash) devices
www.linux-mtd.infradead.org

Microcross, Incorporated
www.microcross.com

uClibc static and shared C library
www.uclibc.org